Yaddes — Yet Another Distributed Discrete Event Simulator: User Manual

Bruno R. Preiss

Ian D. MacIntyre

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 1.
Introduction

The Yaddes system is a tool for constructing discrete event simulations. The principle features of the Yaddes system are:

- the Yaddes simulation specification language and compiler,
- libraries that support various time synchronization methods with extensive built-in trace and debug support,
- a pseudo-random number generator package that supports multiple, independent pseudo-random number streams, and
- a distributed statistics collection and reporting package.

The Yaddes user prepares a specification of the desired simulation. Yaddes then compiles the specification into a collection of C language subroutines. These subroutines are then compiled using the C compiler and then linked to a synchronization method library to form a complete program that performs the desired simulation.

The advantage of the Yaddes system over other discrete event simulation packages is that it uses a modelling approach that supports several different synchronization methods. In particular, the synchronization methods currently provided are:

- sequential (event-list driven) simulation,
- distributed simulation using multiple event lists,
- conservative (Chandy-Misra) distributed simulation, and
- optimistic (virtual-time-based) distributed simulation.

The Yaddes user need not be concerned with the synchronization method used. In fact, every Yaddes specification can be executed using any method merely by linking to the appropriate library. Provided that the specifications are coded properly, the results of a simulation are independent of the synchronization method used.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 2.
Computer Simulation and Modelling

The purpose of computer simulation is to gain insight into the behaviour of an existing or imagined real-world system. Simulation is used both when the real-world system is too complex for mathematical analysis and to validate mathematical analyses when such analyses are possible.

The modelling approach used in the Yaddes system is based on Chandy-Misra distributed discrete event simulation. The real-world system is modelled by a collection of *physical processes* (PPs) that periodically exchange information. This exchange of information takes place at discrete points in time. Every instant at which one PP provides information to another is called an *event*.

The computer simulation of the real-world system is obtained by constructing a computer program in which the behaviour of each PP is mimicked by a *logical process* (LP). The exchange of information by the PPs is mimicked in the simulation by the exchange of messages by the LPs. Since the computer simulation does not execute in real time, each LP has its own notion of time and each message is tagged with the time of the corresponding real-world event.

The Yaddes system provides a language and compiler that aids in the specification of the behaviour of LPs. An LP in the Yaddes system is a general state machine. A general state machine has an arbitrary (finite) number of inputs and outputs and a (finite, albeit possibly large) set of states. The state machine is driven by the occurrence of *event combinations*. **An event combination is a collection of one or more input events having the same time stamp.** In response to an event combination, an LP may change its state and produce zero or more output events on each of its outputs. The Yaddes specification language is used to specify the state of an LP and to associate programs with event combinations. (The programs are written as sequences of C language statements.)

In the Yaddes system, the connections between LPs are static. Each input of every LP must be connected to the output of some other LP. That is, LPs require unity fan-in. The output of an LP may be connected to zero or more LPs. That is, LPs may have arbitrary fan-out. The Yaddes specification language provides a means for enumerating the connections between LPs.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 3.
Synchronization Methods

The Yaddes system currently supports four different synchronization methods. (The methods currently supported are described in the following sections.) However, the Yaddes system hides the details of the underlying synchronization method so that the user need only be concerned with the specification, not the implementation of the simulation. There are two important advantages of the ability to support different synchronization methods. First, by executing the same specification using different synchronization methods, it is possible to directly and quantitatively compare the performance of the synchronization methods. Second, the user can change the synchronization method used without having to recode the simulation specifications. In this way, the most efficient synchronization method can be chosen experimentally.

3.1. Sequential Simulation

The sequential simulation library uses the traditional discrete event simulation method. A single data structure, called the *event list*, is used to hold future events. Future events are sorted by time. The basic execution cycle involves removing events from the event list, forming event combinations, and causing the appropriate LPs to perform the action associated with the given event combination. When an action causes output events, those events are inserted into the event list.

3.2. Distributed Simulation using Multiple Event Lists

This method is a simple extension of the sequential simulation method for execution on a multiprocessor. In this method, each processor has its own event list. In addition, one processor (the scheduler) has special status and acts to coordinate the other processors. The basic execution cycle is somewhat more complex in order to guarantee correct execution on the multiprocessor.

First, each processor sends a message to the scheduler indicating the simulation time of the next event on its event list. The scheduler selects the minimum next event time and sends a message to all the processors containing this value. Each processor having this minimum value removes events from its event list, forms event combinations, and invokes the appropriate LP's actions. When an action causes an output event, that event is either inserted into the local future event list, or a message is sent to a remote processor requesting that it insert an event into its future event list. When a processor is finished executing all the actions for a given value of simulation time, it sends a completion message to all its successors indicating that it is done. Finally, the processor waits until it receives a completion message from all its predecessors. At this point the execution cycle is complete and may begin again.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

In Yaddes, each LP is statically assigned to a processor. This assignment is specified in the Yaddes source. Since the assignment is static, each LP knows a priori whether to insert output events into its own future event list or to send a message to another processor.

3.3. Conservative Distributed Simulation

In conservative (Chandy-Misra) distributed simulation each LP runs as a separate task. Task may run on the same or on different processors. The number of tasks need not be the same as the number of processors.

The basic execution cycle begins when a task receives a message. The task buffers messages until an event combination can be formed. (An event combination with simulation time t can only be formed when a task has received an event message for each of its inputs having time $t' \geq t$.) When an event combination is formed, the LP is activated. If an LP causes an output event, it sends event messages to the appropriate tasks.

This synchronization method has the potential for deadlock. We have not yet implemented any deadlock detection/recovery scheme. We require the simulation programmer to explicitly avoid deadlock.

3.4. Optimistic Distributed Simulation

Our implementation of optimistic distributed simulation is based on the Time Warp Operating System. Each LP runs as a separate task. Tasks exchange messages containing events.

The basic execution cycle begins when a task receives a message. When a message arrives, there are two possibilities — its time stamp is either before or after the current (local) value of simulation time. If its time stamp is after the current time, an input event combination is formed and the LP is activated. If its time stamp is less than or equal to the current time, the task backs up to the time immediately prior to the incoming message. This backing-up is facilitated by an elaborate checkpointing mechanism that allows an earlier state of the LP to be recovered. Essentially, an earlier state is restored, input event combinations are rescheduled, and output events are cancelled by sending *antimessages*.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 4.
Example: Exclusive-OR Circuit

An example to illustrate the appearance of a Yaddes specification is given below. The system in this example is a logic network consisting of four NAND gates that implement the exclusive-OR function as shown in Figure. 4.1.

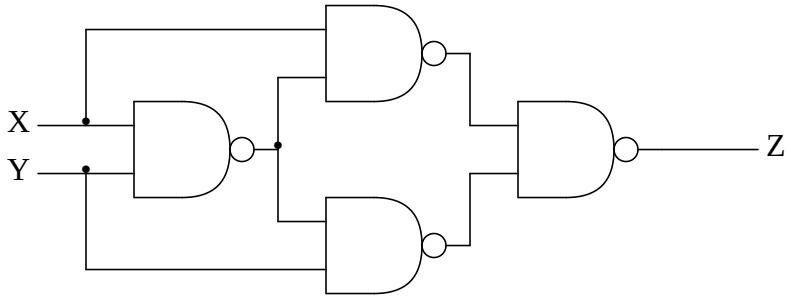

Figure. 4.1. Exclusive-OR Circuit

```
preamble {
#include <stdio.h>

#define NAND(X,Y)  (~((X)&(Y)))

}

constant gateDelay = 10

model TwoInputNand
    inputs in[2]
    outputs out
    state
    {
        int input[2];
    }
    initial state { 0, 0 }
    action initial and not (in[0 or .. 1] or final)
    {
        $output ($out, $time + $gateDelay, NAND ($state->input[0],
            $state->input[1]));
    }
    action in[0] and not (in[1] or initial or final)
```


Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
    {
        $state->input[0] = $event [$in[0]];
        $output ($out, $time + $gateDelay, NAND ($state->input[0],
            $state->input[1]));
    }
    action in[1] and not (in[0] or initial or final)
    {
        $state->input[1] = $event [$in[1]];
        $output ($out, $time + $gateDelay, NAND ($state->input[0],
            $state->input[1]));
    }
    action in[0 and .. 1] and not (initial or final)
    {
        $state->input[0] = $event [$in[0]];
        $state->input[1] = $event [$in[1]];
        $output ($out, $time + $gateDelay, NAND ($state->input[0],
            $state->input[1]));
    }
end model

model ReadFromFile
    inputs none
    outputs out
    state
    {
        char *iname;
        FILE *ifp;
    }
    initial state { NULL, NULL }
    action initial and not final
    {
        int time, event;

        if (($state->ifp = fopen ($state->iname, "r"))
            == NULL)
        {
            (void) fprintf (stderr, "can't open %s\n",
                $state->iname);
            (void) exit (1);
        }
        while (fscanf ($state->ifp, "%d%d", &time, &event) == 2)
        {
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
            $output ($out, time, event);
        }
    }
end model

model WriteToFile
    inputs in
    outputs none
    state
    {
        char *oname;
        FILE *ofp;
    }
    initial state { NULL, NULL }
    action initial and not (in or final)
    {
        if (($state->ofp = fopen ($state->oname, "w")) == NULL)
        {
            (void) fprintf (stderr, "can't open %s\n",
                $state->oname);
            (void) exit (1);
        }
    }
    action in
    {
        (void) fprintf ($state->ofp, "%d %d\n", $time,
            $event [$in]);
    }
end model

process X : ReadFromFile {"file1.in", NULL}
process Y : ReadFromFile {"file2.in", NULL}
process Gate[4] : TwoInputNand
process Z : WriteToFile  {"file.out", NULL}

connect X.out to Gate[0].in[0], Gate[1].in[0]
connect Y.out to Gate[0].in[1], Gate[2].in[1]
connect Gate[0].out to Gate[1].in[1], Gate[2].in[0]
connect Gate[1..2].out to Gate[3].in[0..1]
connect Gate[3].out to Z.in
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

The model called `TwoInputNand` describes the behavior of a NAND gate. `TwoInputNand` has two inputs called `in[0]` and `in[1]` and a single output called `out`. `TwoInputNand` has two state variables called `input[0]` and `input[1]`. These variables record the logic level (0 or 1) on the corresponding inputs.

The `TwoInputNand` model has an `initial` action and three event combination actions. The event combinations are (i) an event occurred on input `in[0]`, (ii) an event occurred on input `in[1]`, and (iii) simultaneous events occurred on both `in[0]` and `in[1]`. Each of these actions updates the appropriate fields of the state and generates an output event representing the logical NAND of the inputs after a gate delay of 10 time units.

The model called `ReadFromFile` represents an LP that reads input events for the simulation from a file. The file is assumed to contain pairs of integers. The first integer represents the time of an event and the second integer is either 0 or 1 representing a logic level. The `ReadFromFile` model only has an `initial` action. It opens a file whose name is obtained from the state of the LP and then reads input events until the end-of-file is reached.

The model called `WriteToFile` represents an LP that writes output events from the simulation into a file. The file format is the same as the input file format described above. In its `initial` action the `WriteToFile` model opens a file whose name is obtained from the state of the LP. The `WriteToFile` model has a single input. Whenever an input event arrives, the `WriteToFile` model prints a pair of integers representing the time and event into the output file.

A total of seven LPs are declared in the specification above. In addition to the four NAND gates, there are two instances of the `ReadFromFile` model and an instance of the `WriteToFile` model.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 5.
The Yaddes Specification Language

The general Yaddes specification file contains specifications for models, (logical) processes, connections, and procedures. In addition, the file may also contain a C language preamble and postamble. The appearance of a specification file is

```
preamble
{
        optional C code
}
model specifications
process specifications
connection specifications
procedure specifications
postamble
{
        optional C code
}
```

All specifications are optional (including the `preamble` and `postamble`). In order for there to be any connection specifications, there must be at least one process specification. In order for there to be any process specifications, there must be at least one model specification. The order of the specifications does not matter except that any objects used in a specification must be declared in an earlier specification.

5.1. Model Specifications

A model specification describes a type of LP. A simulation will typically have many different types of LP. All LPs are instantiations of models. For example, if the system being simulated is a logic circuit, then the models might describe *and*, *or*, and *not* gates. The logic circuit can be constructed by instantiating these models to represent the actual gates in the circuit.

The format of a model specification is

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`model` *model name*
input statement
output statement
state statement
initial state statement
action statements
`end model`

All the statements are mandatory except for the *action statements*. Hence the minimal model looks like

`model` *model name*
input statement
output statement
state statement
initial state statement
`end model`

The order of the statements within the model specification does not matter except that any objects used in a statement must be declared in an earlier statement.

5.1.1. Inputs

The `inputs` statement is used to specify the number and names of the inputs of a model. Each input and output of a given model must have a distinct name. (The names of inputs and outputs of different models need not be distinct.)

For example, the statement

```
inputs a, b [3], c [3..5]
```

declares seven inputs: `a`, `b[0]`, `b[1]`, `b[2]`, `c[3]`, `c[4]`, and `c[5]`. Note that there are two forms of vector declaration. The first form requires a single expression. This expression is taken as the size of the vector. In this case, the first element is numbered `0`. The second form requires two expressions. These expressions are taken as the lower and upper subscripts (respectively) of the vector. The array subscript expressions may be numeric values, symbolic constants or arbitrary manifest expressions. (See Constants and Manifest Expressions.)

The statement

```
inputs none
```

specifies that this model has no inputs. (I.e., it is a source of events.) Note, it is not possible to name an input `none`.

A model can have from zero up to 30 inputs although the realistic maximum number is considerably less. This limitation is imposed by the number of case labels generated by Yaddes for the model action statements. In the worst case, the number of case labels can grow to 2^{n+2}

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

labels for *n* inputs. A practical limit is around eight to ten inputs.

5.1.2. Outputs

The `outputs` statement is used to specify the number and names of the outputs of a model. A model can have zero or more outputs. (There is no inherent limit on the number of outputs that a model can have.) Each input and output of a given model must have a distinct name. (The names of inputs and outputs of different models need not be distinct.)

For example, the statement

```
outputs a, b [3], c [3..5]
```

declares seven outputs: `a`, `b[0]`, `b[1]`, `b[2]`, `c[3]`, `c[4]`, and `c[5]`. Note that there are two forms of vector declaration. The first form requires a single expression. This expression is taken as the size of the vector. In this case, the first element is numbered `0`. The second form requires two expressions. These expressions are taken as the lower and upper subscripts (respectively) of the vector. The array expressions may be numeric values, symbolic constants or arbitrary manifest expressions. (See Constants and Manifest Expressions.)

The statement

```
outputs none
```

specifies that this model has no outputs. (I.e., it is a sink of events.) Note, it is not possible to have an output named `none`.

5.1.3. State

The `state` statement is used to declare the type of the state of a model. Each instantiation of a model (i.e., LP) has its own unique state with the specified type. The syntax of this type declaration is that of a C language `struct` declaration.

For example, the statement

```
state
{
    int x;
    int y;
    int z;
}
```

specifies that the state of the model consists of three integers `x`, `y`, and `z`. Any valid C language structure field declaration is allowed between the braces. (Due to a quirk of C, empty `struct`s are not allowed. Thus, the model state type must have at least one field.)

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.1.4. Initial State

The `initial state` field is used to specify the value to which the state of each instantiation of this model is initialized. The syntax of this statement is that of a C language structure initializer. For example, the statement

```
initial state { 0, 0, 0 }
```

specifies that the first three field of the state are initialized to zero. Any valid C language structure field value is allowed between the braces. (Note that the C language compiler does not check that the number of values between the braces matches the number fields in the corresponding structure type declaration. The programmer must exercise care when initializing structures.)

5.1.5. Actions

Actions are C language program fragments that are invoked during the course of a simulation. Actions can be triggered in a number of ways. Specifically, actions can be triggered before simulation begins, before simulation ends, whenever simulated time advances, and in response to combinations of events.

5.1.5.1. General Event Combination Actions

A combination of events is a set of events that occur at the same simulation time. For example, the statement

```
action a and b and c
{
    printf ("hello world");
}
```

specifies that whenever events occur on inputs `a`, `b`, and `c` at the same simulation time, the C language print statement is invoked.

An action statement is composed of a Boolean expression, to control the invocation of the action, and a sequence of C language statements. Any valid C language statement sequence is allowed between the braces. In addition, certain special variables and constants are available for use in the statement sequence. (See section 5.5 and 5.6.)

For example, the statement

```
action a and not (initial or final or b or c)
{
    printf ("hello world");
}
```

specifies that whenever an event occurs on only input `a`, the C language print statement is invoked. This assumes that the model has no inputs other than `a`, `b`, and `c`. If this is not the case, then the print statement is invoked when an event occurs on input `a`, no event occurs on

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

inputs `b` and `c`, and any combination of events occur on the other inputs.

The action selector expressions may contain the Boolean operators `and`, `or`, and `not`. Parentheses may be used to group operands and to override the natural precedence of the Boolean operators. When input vectors are used, it is sometimes desirable to compute a Boolean operation over all of the elements of the vectors. The form `a [0 or .. 2]` is equivalent to `a[0] or a[1] or a[2]`. The form `a [0 and .. 2]` is equivalent to `a[0] and a[1] and a[2]`.

5.1.5.2. Initial Action

The `action initial` statement is used to specify an action that is invoked exactly once at simulation time zero. For example, the statement

```
action initial and not ( ... )
{
    printf ("hello world");
}
```

specifies that the C language print statement is to be invoked at simulation time zero.

PLEASE NOTE: The initial action occurs at time 0. Since Yaddes requires that no two actions can be invoked with the same time, and since ALL LPs are given the opportunity to perform an initial action, no events may be scheduled at time 0.

5.1.5.3. Final Action

The `action final` statement is used to specify an action that is invoked exactly once immediately prior to the end of the simulation. For example, the statement

```
action final and not ( ... )
{
    printf ("hello world");
}
```

specifies that the C language print statement is to be invoked immediately prior to ending the simulation.

5.1.5.4. No Event Action

The `action nothing` statement is used to specify an action that is invoked when no events have occurred, but simulation time has advanced. This situation will only occur when using the conservative synchronization method together with the special functions `$ignore` and `$deactivate`. For example, the statement

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
action nothing
{
    printf ("hello world");
}
```

specifies that the C language print statement is to be invoked whenever no events have occurred, but simulation time has advanced. (See section 5.6.)

5.1.5.5. Some Event Action

The `action something` statement is used to specify an action that is invoked when an event combination with at least one event occurs. For example, the statement

```
action something
{
    printf ("hello world");
}
```

specifies that the C language print statement is to be invoked when an event combination involving at least one event occurs.

5.1.5.6. Any Event Action

The `action anything` statement is used to specify an action that is invoked when an event combination occurs or simulation time advances. It is equivalent to `nothing or something`. For example, the statement

```
action anything
{
    printf ("hello world");
}
```

specifies that the C language print statement is to be invoked when an event combination involving at least one input event occurs or when simulation time has advanced but no events have occurred.

5.1.5.7. Default Action

The `action default` statement is used to specify an action that is invoked when an event combination has occurred for which no other action has been explicitly declared. For example, the statement

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
action default
{
    printf ("hello world");
}
```

specifies that the C language print statement is to be invoked when an event combination has occurred for which no other action has been explicitly declared.

If a default action is not specified, a default default action is invoked. The default default action simply prints a warning message that specifies the event combination. (Note, warnings can be suppressed. See section 5.9.2.)

5.2. Process Specifications

A `process` statement declares an instantiation of a model. Each such instantiation executes as a separate LP. Each such LP has its own distinct state. For example, the statements

```
process a : node
process b [3] : node
process c [3..5] : node
```

specify seven LPs called `a`, `b[0]`, `b[1]`, `b[2]`, `c[3]`, `c[4]`, and `c[5]`. These LPs are all instances of the model called `node`. Each of these LPs has its own distinct state. The type of this state is that specified in the `node` model. Each LP has the set of inputs and a set of outputs that is specified in the `node` model.

Note that there are two forms of vector declaration. The first form requires a single expression. This expression is taken as the size of the vector. In this case, the first element is numbered `0`. The second form requires two expressions. These expressions are taken as the lower and upper subscripts (respectively) of the vector.

The array expressions may be numeric values, symbolic constants or arbitrary manifest expressions. (See Constants and Manifest Expressions.)

Yaddes provides the facility for assigning an initial state to an LP in the process declaration statement. For example, the statements

```
process a : gate
process b : gate { 20 }
```

are both valid Yaddes process declarations. The first form above assigns the default initial state (obtained from the model declaration) to the LP `a`. The second form assigns the specified initial state to the LP `b`. Thus, it is possible to assign a unique initial state to each and every LP. Any valid C language structure field value is allowed between the braces. (Note that the C language compiler does not check that the number of values between the braces matches the number fields in the corresponding structure type declaration. The programmer must exercise care when initializing structures.)

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

In some execution environments there may be several processors on which the LPs may execute. In this case, each LP must be assigned to a processor. Processor selection is performed as in the following example.

```
process a [3] : node
process b [3] on 1 : node
process c [3] on 0..2 : node
process d [7] on 0..2 : node
```

The first form above declares three LPs, `a[0]`, `a[1]`, and `a[2]`, and assigns them (by default) to processor 0.

The second form above declares three LPs, `b[0]`, `b[1]`, and `b[2]`, and assigns them all to processor 1.

The third form above declares three LPs, `c[0]`, `c[1]`, and `c[2]`, and assigns them to processors 0, 1, and 2, respectively.

The fourth form above declares seven LPs, `d[0]`, `d[1]`, `d[2]`, `d[3]`, `d[4]`, `d[5]`, and `d[6]`, and assigns them to processors 0, 1, 2, 0, 1, 2, and 0, respectively. This fourth example illustrates how Yaddes handles a mismatch between the size of the process vector and the size of the range in the `on` clause of the process declaration.

Processors are always numbered starting with 0. There may be any number of processors. The number of processors is obtained by taking the largest processor number specified and adding 1. Note that it is possible to have a processor unused simply by not assigning LPs to it. If an LP is not explicitly assigned to a processor, it is assigned to processor number 0 by default.

5.3. Connection Specifications

The `connect` statement is used to establish connections between the inputs and outputs of LPs. Each output of a LP can be connected to zero, one or more inputs. (I.e., fan-out is allowed.) Each input of a LP can be connect to at most one output. (I.e., fan-in is not allowed.) The inputs and outputs of LPs are specified in the form *process name* . *port name*, where *process name* is the name of an LP, and *port name* is the name of an input or output of that LP. For example, the statement

```
connect Gate1.out to Gate2.in0
```

specifies that the output `out` from the LP named `Gate1` is connected to the input `in0` of the LP named `Gate2`. An output port with fan-out can be specified by connecting to more than one input port. For example, the statement

```
connect Gate1.out to Gate2.in0, Gate2.in0
```

specifies a fan-out of two.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Yaddes provides support for the declaration of arrays of connections. For example, the following are valid declarations in Yaddes:

```
connect a.out[0..3] to b.in[0..3], c.in[0..3]
connect d[0..3].selfOut to d[0..3].selfIn
connect e[0..3].out[0..2] to f[0..2].in[0..3]
```

The first form above is equivalent the to following sequence of connection declarations:

```
connect a.out[0] to b.in[0], c.in[0]
connect a.out[1] to b.in[1], c.in[1]
connect a.out[2] to b.in[2], c.in[2]
connect a.out[3] to b.in[3], c.in[3]
```

The second form above is equivalent to the following sequence of connection declarations:

```
connect d[0].selfOut to d[0].selfIn
connect d[1].selfOut to d[1].selfIn
connect d[2].selfOut to d[2].selfIn
connect d[3].selfOut to d[3].selfIn
```

The third form above is equivalent the following sequence of connection declarations:

```
connect e[0].out[0] to f[0].in[0]
connect e[0].out[1] to f[0].in[1]
connect e[0].out[2] to f[0].in[2]
connect e[1].out[0] to f[0].in[3]
connect e[1].out[1] to f[1].in[0]
connect e[1].out[2] to f[1].in[1]
connect e[2].out[0] to f[1].in[2]
connect e[2].out[1] to f[1].in[3]
connect e[2].out[2] to f[2].in[0]
connect e[3].out[0] to f[2].in[1]
connect e[3].out[1] to f[2].in[2]
connect e[3].out[2] to f[2].in[3]
```

This last example illustrates the following points. First, if connection declaration involves both an array of processes and an array of inputs or outputs, the total number of output ports involved (i.e., number of processes times the number of outputs) must be the same as the total number of input ports involved (i.e., the number of processes times the number of inputs). Second, the connections are established by looping over the input/output ports and then over the processes.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.4. Procedure Specifications

5.4.1. Initial procedure

The initial procedure declaration allows the Yaddes user to specify the name of a C subroutine that is to be called before the simulation begins. For example, the statement

```
procedure SetUp : initial
```

specifies that the name of the C language subroutine that is to be invoked before the simulation begins is `SetUp`. It is the responsibility of the Yaddes user to provide this routine. (E.g., it can be declared in the preamble or postamble or in a separate file.)

On a multiprocessor, the initial procedure will be executed once on each processor.

By default, the initial procedure is a call to `DefaultInitial`. If the initial procedure is redefined, and the user still wishes to call `DefaultInitial`, then the call must be made explicitly from the user-defined final procedure.

On a multiprocessor, one activation of `DefaultInitial` will occur in each processor.

In the current implementation of Yaddes, the procedure `DefaultInitial` does nothing.

The initial procedure is invoked by the run-time kernel as follows:

```
SetUp (argc, argv);
```

`argc` and `argv` are declared in the normal way for C programs. `argc is an integer specifying the number of command-line arguments. argv is an array of pointers to character strings, one for each command-line argument. In this way, the simulation user can access information from the command-line used to invoke the simulation.`

N.B., the command line syntax used must be compatible with the Yaddes run-time support routines. Some arguments are interpreted by the support routines to control the simulation. All command-line arguments are passed to the initial procedure. Thus, the initial procedure must accept (and ignore if appropriate) all Yaddes-related command-line arguments.

5.4.2. Final procedure

The final procedure declaration allows the Yaddes user to specify the name of a C subroutine that is to be called after the simulation ends. For example, the statement

```
procedure CleanUp : final
```

specifies that the name of the C language subroutine that is to be invoked after the simulation ends is `CleanUp`. It is the responsibility of the Yaddes user to provide this routine. (E.g., it can be declared in the preamble or postamble or in a separate file.)

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

By default, the final procedure is a call to `DefaultFinal`. (See chapter 7.) If the final procedure is redefined, and the user still wishes to call `DefaultFinal`, then the call must be made explicitly from the user-defined final procedure.

On a multiprocessor, the final procedure will be executed once on each processor. On a multiprocessor, one activation of `DefaultFinal` will occur in each processor.

5.5. Predefined Variables and Constants

The Yaddes user has access to certain predefined variables and constants inside the C language portions of actions. The names of these variables and constants all begin with the symbol `$`.

5.5.1. Constants

Each of the inputs and outputs of a model is assigned an integer value. For example, if a model has the following input specification

```
inputs a, b, c[3]
```

then the integer constants `$a`, `$b`, `$c[0]`, `$c[1]`, and `$c[2]` are defined. (In the current version of Yaddes the values will be 0, 1, 2, 3, and 4, respectively.)

Similarly, if a model has the following output specification

```
outputs a, b, c[3]
```

then the integer constants `$a`, `$b`, `$c[0]`, `$c[1]`, and `$c[2]` are defined. (In the current version of Yaddes the values will be 0, 1, 2, 3, and 4, respectively.)

Of note is the ability to use non-manifest expressions as subscripts for input and output vectors. For example, the C code may refer to the constant:

```
$c [i+1]
```

where `c` is an input or output vector (defined as above), and `i+1` is a non-manifest expression (i.e., one for which the value is not known until run time).

For information on the use of these values, see section 5.6.

5.5.2. Variables

5.5.2.1. `$event`

The `$event` variable is an array of integers. Its declaration is equivalent to the following:

```
int $event [];
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

When an action is invoked, the event array contains the events that triggered that action. For example, if the action trigger is

```
action a and b[1] and not c
```

then `$event[$a]` contains the event that occurred on input `a` and `$event[$b[1]]` contains the event that occurred on input `b[1]`. The value in `$event[$c]` is undefined. (In the current version undefined values are always 0. This may not be the case in future versions of Yaddes. The Yaddes user should use the predefined function `$eventoccurred` (see section 5.6.3) to determine if the `$event` array entry is defined for a particular input.)

5.5.2.2. `$name`

The `$name` variable is a pointer to a character string that contains the name of the process (not of the model). Its declaration is equivalent to the following:

```
char * $name;
```

This can be used in a print statement to aid debugging.

5.5.2.3. `$pid`

The `$pid` variable is an integer that contains the process ID of the LP. Every LP is assigned a unique ID by the Yaddes compiler. Its declaration is equivalent to the following:

```
int $pid;
```

This quantity is typically used to provide a seed to initialize a random number generator.

5.5.2.4. `$state`

The `$state` variable is a pointer to a structure whose type is that declared in the `state` statement of the model specification. Using this pointer, the Yaddes user can alter the current state of a process.

Advanced Yaddes users sometimes want to pass the variable `$state` to a subroutine. In this case, one needs to know how the variable is declared. The `$state` variable points to a different type of structure within each model. The type name of the structure for a particular model is the concatenation of `xx`, followed by the model name, followed by `STATE`. For example, for the model named `TwoInputNand`, the declaration of the variable `$state` is equivalent to the following

```
xxTwoInputNandSTATE * $state;
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.5.2.5. `$time`

The `$time` variable is an integer. Its declaration is equivalent to the following:

```
int $time;
```

It contains the current value of simulation time.

5.6. Predefined Functions

The Yaddes user has access to certain predefined functions inside the C language portions of actions. These functions are used to cause events to occur on the outputs of processes and to control the simulation. The names of these functions all begin with the symbol $.

5.6.1. `$activate`

The `$activate` function is used in conjunction with the `$deactivate` function. The declaration of this function is equivalent to the following:

```
void $activate (port)
int port;
```

The purpose of this function is to turn on an input to a process that has been previously turned off by the `$deactivate` function. For example, the statement

```
$activate ($a);
```

turns input `a` on.

5.6.2. `$deactivate`

The `$deactivate` function is used to turn off an input to a process. The declaration of this function is equivalent to the following:

```
void $deactivate (port)
int port;
```

For example, the statement

```
$deactivate ($a);
```

turns off input `a`. When an input is turned off, it means that no events are expected on that input. Should any events occur on a `$deactivate`d input, warning messages will be printed. (Warnings can be suppressed. See section 5.9.2.)

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.6.3. $eventoccurred

The $eventoccurred function is used to test whether an input has an event on it. The declaration of this function is equivalent to the following:

```
int $eventoccurred (port)
int port;
```

This function returns zero if no event has occurred on the specified input. It returns a nonzero result if an event has occurred.

This function is useful in actions associated with multiple event combinations. The following example illustrates the use of this function in a multiple event combination action:

```
action a and (b or not b)
{
    if ($eventoccurred ($b))
        printf ("hello world");
}
```

5.6.4. $ignore

The $ignore function is used to cause an input to a process to be turned off until simulation time has advanced past a specified time value. The declaration of this function is equivalent to the following:

```
void $ignore (port, time)
int port;
int time;
```

For example, the statement

```
$ignore ($a, $time + 10);
```

specifies that input a is to be turned off until simulation time has advanced to the current simulation time plus ten time units. When an input is turned off, it means that no events are expected on that input. Should any events occur on an $ignored input, warning messages will be printed (unless warnings are suppressed). (See section 5.9.2.)

5.6.5. $output

The $output function is used to cause an event to occur on an output of a process. The declaration of this function is equivalent to the following:

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
void $output (port, time, event)
int port;
int time;
int event;
```

For example, the statement

```
$output ($a, $time + 10, 57);
```

specifies that an event is to occur on output `a` at current simulation time plus ten time units. The event value is `57`. On uniprocessor simulations, it is possible to pass a pointer as the event value. This is not possible on distributed simulations.

5.6.6. `$nulloutput`

The `$nulloutput` function is similar to the `$output` function. Its declaration is equivalent to the following:

```
void $nulloutput (port, time, event)
int port;
int time;
int event;
```

This function is used in the same manner as `$output` to cause events to occur on the outputs of a process. The semantics of this function are identical to those of the `$output` function. Its purpose is to identify those events that are required to avoid deadlock when using the Chandy-Misra distributed discrete event simulation mechanism. Under other execution mechanisms, `$nulloutput` is a no-op.

Note, there is no guarantee that the contents of a `$nulloutput` message will ever arrive at the destination LP. For example, if a message with a later time stamp ever ‘‘catches up’’ with an null message with an earlier time stamp, then the null message may be discarded if null message cancellation is enabled. (See section 5.9.2.)

5.6.7. `$settimelimit`

The `$settimelimit` function is used to specify the simulation time at which an LP is to terminate its execution. The declaration of this function is equivalent to the following:

```
void $settimelimit (time)
int time;
```

For example, the statement

```
$settimelimit (1000);
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

will cause the LP to terminate execution when simulated time advances past time `1000`.

This function is usually called in the initial action of a model. Note that when a process terminates, its final action is invoked. In effect, this function schedules the final action at a specified time.

The precise semantics of this function depends on the synchronization method used. In some synchronization methods, each LP has its own notion of time. In this case, `$settimelimit` only affects the LP in which it is invoked. In synch methods using global time synchronization, a global time limit is used. In this case, the time limit is the minimum value of time specified in all of the calls to `$settimelimit`.

5.7. Constants and Manifest Expressions

Yaddes supports the declaration of symbolic constants. For example, the following statement declares the symbol `shoesize` and assigns to it the value 57:

```
constant shoesize = 57
```

Yaddes supports arbitrary, integer-valued, manifest expressions wherever a numeric value is required. (A manifest expression is one whose value can be evaluated during translation.) A manifest expression may contain integers, symbolic constants, and the following operators: `'+'`, `'-'`, `'*'`, `'/'`, and `'%'`. For example, the following statement declares the symbol `bar` and assigns to it the value 59:

```
constant hatsize = shoesize + (3 * 4) / 5
```

Yaddes constants can be accessed inside the C language portions of the specification file. In order to access a Yaddes constant in the C code, its name must be prefixed with the symbol `$`. For example, the following example shows how Yaddes constants can be used:

```
constant numberOfInputs = 2
inputs in [numberOfInputs]
action default
{
    printf ("%d0, $numberOfInputs);
}
```

Note that the constant must not be prefixed with the symbol `$` when used elsewhere within the Yaddes (non-C code) specifications.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.8. Comments

Yaddes supports embedded comments. Note that there is a distinction between the C code portion of a Yaddes specification and the non-C portion. Within the C code portions, the usual C rules for comments apply. Within the non-C portions two styles of comments are supported: First, comments may be contained within the delimiters `/*` and `*/`. These delimiters may be nested. Second, the symbol # introduces a comment that continues until the end of the line.

5.9. Compiling and Running the Simulation

5.9.1. Compiling a Yaddes Language Specification

The Yaddes program transforms a specification file into a C language file. The Yaddes program can be invoked in two ways. For example,

`yaddes2` *input file*

transforms the specifications in the *input file*. If the input file name ends in `.y`, the output file name is constructed by replacing the `.y` with `.c`. Otherwise `.c` is added to the end of the input file name to construct the output file name.

If Yaddes is invoked without an argument, it reads the standard input file and writes the standard output file. For example,

`yaddes2 <` *input file* `>` *output file*

Note, the name under which the Yaddes compiler is installed may vary from one installation to another. In this example, it is assumed that the program is installed as `yaddes2`.

5.9.1.1. Separate Compilation

If a simulation contains many models, it may become convenient to separate the models into different input files. Yaddes supports a form of separate compilation in that model specifications can appear in separate files. However, all the process, connection, and procedure declarations must appear in one file.

Yaddes can extract model information directly from the file containing the model declaration. For example, consider the following Yaddes statement:

```
import modelA, modelB from "file.y"
```

This statement declares that the file named `file.y` contains the declaration of two models, named `modelA` and `modelB`.

Yaddes will extract the needed information from the specified file. For this to be possible, the file must not contain syntax errors. In addition, it is only possible to import models and constants. Finally, if the file contains process or connection specifications, those are ignored.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.9.1.2. Synchronization Method Libraries

Once the C language program file (or files) has been produced, the C compiler can be used to create an executable object module. In order to create this module, the C files must be linked to a synchronization method library. There are currently 18 such libraries available. The libraries are summarized in the following tables:

Table 1. Portable (Uniprocessor) Libraries

library	description
event-list.a	sequential (event-list driven) simulation
multi-list.a	distributed simulation using multiple event lists
chandy-misra.a	conservative distributed simulation
virtual-time.a	optimistic distributed simulation

Table 2. Portable (Uniprocessor) Libraries for the IBM PC

library	description
event_li.lib	sequential (event-list driven) simulation
multi_li.lib	distributed simulation using multiple event lists
chandy_m.lib	conservative distributed simulation
virtual_.lib	optimistic distributed simulation

Table 3. Apollo Distributed Simulation Libraries

library	description
dist-multi-list.a	distributed simulation using multiple event lists (runs on a network of Apollo workstations)
dist-chandy-misra.a	conservative distributed simulation (runs on a network of Apollo workstations)
dist-virtual-time.a	optimistic distributed simulation (runs on a network of Apollo workstations)

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Table 4. VAX Simulated Multiprocessor Libraries

library	description
multi-list.vax.a	distributed simulation using multiple event lists (simulates a multiprocessor by multitasking under a single unix process)
chandy-misra.vax.a	conservative distributed simulation (simulates a multiprocessor by multitasking under a single unix process)
virtual-time.vax.a	optimistic distributed simulation (simulates a multiprocessor by multitasking under a single unix process)

Table 5. Transputer T800 (Uniprocessor) Simulation Libraries

library	description
event_li.lib	sequential (event-list driven) simulation (runs on a single T800 Transputer processor)
multi_li.lib	distribute simulation using multiple event lists (runs on a single T800 Transputer processor)
chandy_m.lib	conservative distributed simulation (runs on a single T800 Transputer processor)
virtual_.lib	optimistic distributed simulation (runs on a single T800 Transputer processor)

Table 6. Transputer T414 Distributed Simulation Libraries

library	description
dist_el.lib	sequential (event-list driven) simulation (runs on one node in a network of T414 Transputer processors)
dist_ml.lib	distributed simulation using multiple event lists (runs on a network of T414 Transputer processors)
dist_cm.lib	conservative distributed simulation (runs on a network of T414 Transputer processors)
dist_vt.lib	optimistic distributed simulation (runs on a network of T414 Transputer processors)

The sequence of commands

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
yaddes2 example.y
cc -o example example.c event-list.a
```

produces an executable object module called `example` from the Yaddes Language specification file `example.y`, in this case using the portable event-list simulation library.

5.9.2. Running a Simulation Program

The run-time execution mechanism libraries support a number of command line arguments. These arguments turn on or off the printing of various statistics and debugging messages. All messages are printed on the standard error file. If the executable module is called `example` then the format of the command to invoke the simulation is

`example` *[flag [flag ...]]*

Note that one or more flags may be included on the command line. Some modifications to this procedure are necessary for the distributed simulations. (See the Apollo and Transputer multiprocessor appendices.)

5.9.2.1. `+a`

This flag turns on all forms of output. (Except the help output.)

5.9.2.2. `+h` and `-h`

These flags cause a brief summary of the flags to be printed. The simulation is not run when this flag is specified.

5.9.2.3. `+i`

This flag causes the printing of messages that trace calls to `$ignore`, `$activate`, and `$deactivate`.

5.9.2.4. `+k`

This flag causes the printing of messages that trace the execution of the kernel.

5.9.2.5. `+m`

This flag causes the printing of a summary of message statistics. This summary specifies the number of messages received on each input of every process.

5.9.2.6. `+o`

This flag causes the printing of messages that trace calls to `$output` and `$nulloutput`.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.9.2.7. +p

This flag causes a printing of a post mortem when the simulation terminates. The post mortem lists all the message that have been received by processes, but not yet processed.

5.9.2.8. +s

This flag causes the printing of various statistics gathered by the kernel. These statistics include the number of messages sent and received, the number of actions invoked, and the number of context switches. The statistics produced by the various kernels are described in the appendices.

5.9.2.9. +t

This flag causes the printing of a detailed trace of actions as they are invoked.

5.9.2.10. -w

This flag suppress the printing of warning messages.

5.9.2.11. +nmc

This flag enable null message cancellation in conservatively synchronized distributed simulation. By default, null message cancellation is disabled.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 6.
Pseudo-Random Number Generator Package

The Yaddes system provides support for multiple, independent pseudo random number streams. This allows each process to have its own stream of random numbers that is independent of the other processes. This is essential in order to ensure that the results of the simulation are independent of the underlying simulation mechanism.

6.1. Using the Pseudo-Random Number Generator Package

Pseudo-random number generators require internal state in order to operate. Every time a random number is generated, the internal state is modified. In order to ensure correct operation in the Yaddes environment, this state must be part of the state of the LP. The Yaddes system provides a predefined type, `RNG_TYPE`, and initial value, `RNG_INITIAL_VALUE`, that are to be used in the model state and initial state specifications as shown here:

```
state
{
    RNG_TYPE rng_state;
    ...
}
initial state
{
    RNG_INITIAL_VALUE,
    ...
}
```

Pseudo-random number generations must be initialized before use. The `InitializeRNG` routine is used to initialize the pseudo-random number generator. The declaration of this routine is equivalent to the following:

```
void InitializeRNG (seed, state_ptr)
int seed;
RNG_TYPE *state_ptr;
```

The first argument is an integer used to seed the random number generator. The second argument is a pointer to a random number generator state variable. A convenient place to initialize the pseudo-random number generator in Yaddes is in an initial action as shown here:

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
action initial and not ( ... )
{
    InitializeRNG ($pid, &($state->rng_state));
}
```

If the call to this routine is omitted, then the default initial state, `RNG_INITIAL_STATE`, is used. This state is equivalent to initializing using a seed of 1.

The routine called `Random` is used to generate the actual pseudo-random number sequence. The declaration of this routine is equivalent to the following:

```
int Random (state_ptr)
RNG_TYPE *state_ptr;
```

Every time this routine is invoked it returns a different integer in the range 0 to $2^{31} - 1$. The following statement shows how to invoke the pseudo-random number generator:

```
n = Random (&($state->rng_state));
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 7.
Statistics Gathering and Reporting Package

The Yaddes system provides a set of routines that facilitate the gathering and reporting of statistics. Two types of statistic are provided: population and time averaged. The reporting routines report for each statistic the minimum value, the maximum value, the number of values, the sum of the values, the sum of the squares of the values, the mean, and the standard deviation.

In order to accumulate statistics, they must be incorporated into the state of the process. The Yaddes system provides a predefined type, `STATISTIC_TYPE`, and initial value, `STATISTIC_INITIAL_VALUE`, that are to be used in the model state and initial state specifications as shown here:

```
state
{
    STATISTIC_TYPE stat1;
    STATISTIC_TYPE stat2;
    ...
}
initial state
{
    STATISTIC_INITIAL_VALUE,
    STATISTIC_INITIAL_VALUE,
    ...
}
```

Note that each process may accumulate more than one statistic.

The Yaddes user must initialize the state of each statistic and declare whether the statistic is a population or time-averaged statistic. The routine `DeclareStatistic` is used to initialize a statistic. The declaration of this routine is equivalent to the following:

```
void DeclareStatistic (stat_ptr, type)
STATISTIC_TYPE *stat_ptr;
int type;
```

The first argument is a pointer to a statistic state variable. The second argument specifies the type of the statistic. There are two types currently supported: `POPULATION` and `TIME_AVERAGED`. A convenient place to initialize statistics in Yaddes is in an initial action as shown here:

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
action initial and not ( ... )
{
    DeclareStatistic (&($state->stat1), POPULATION);
    DeclareStatistic (&($state->stat2), TIME_AVERAGED);
}
```

If the call to this routine is omitted, then the default initial state, `STATISTIC_INITIAL_STATE`, is used. This state is equivalent to declaring the statistic to be a population statistic.

The routine `AccumulateStatistic` is used to gather statistics. The declaration of this routine is equivalent to the following:

```
void AccumulateStatistic (stat_ptr, value, time)
STATISTIC_TYPE *stat_ptr;
int value;
int time;
```

The first argument is a pointer to a statistic state variable. The second argument is the new value of the statistic. The third argument is required to be the current value of simulation time. The following statement shows how to accumulate a statistic:

```
int val;

val = ... ;
AccumulateStatistic (&($state->stat1), val, $time);
```

The routine called `ReportStatistic` is used to collect statistics in preparation for printing. The declaration of this routine is equivalent to the following:

```
void ReportStatistic (stat_ptr, label)
STATISTIC_TYPE *stat_ptr;
char *label;
```

The first argument is a pointer to a statistic state variable. The second argument is a pointer to a character string that gives a name for the statistic. All statistics with a given name are collected together — regardless of whether they are generated in the same or in different LPs (see section 7.1). A convenient place to collect statistics in Yaddes is in a final action as shown here:

```
action final and not ( ... )
{
    ReportStatistic (&($state->stat1), "customer age");
    ReportStatistic (&($state->stat2), "customers in queue");
}
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

The routine called `DefaultFinal` is used to print the collected statistics on the standard output file. The declaration of this routine is equivalent to the following:

```
void DefaultFinal ()
```

This routine should the last procedure called By default, the final procedure is automatically declared to be `DefaultFinal` (see section 5.4). If this default is overridden, then the Yaddes user must explicitly call `DefaultFinal` from within the new final procedure.

7.1. Distributed Statistics Collection Capability

In many cases, the user of a simulation wishes to collect and average statistics generated in several different LPs. The `ReportStatistic` routine uses its second argument, the name of the statistic, to automatically collect and average statistics generated in different LPs. All statistics with the same name will be collected together and the aggregate statistics reported.

On a multiprocessor, this requires the collection of statistics from different processors. The run-time libraries do this automatically.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 8.
Avoiding deadlock when using Conservative Synchronization

A problem with the conservative synchronization methods is the possibility for deadlock. To understand how deadlock can arise, consider the system of LPs shown schematically in Figure. 8.1. The system consists of two LPs — X and Y. Assume that LP Y only outputs events on its output arc in response to an event arriving on its input arc. This means that LP X only receives input events from Y if it has sent an event to Y at some earlier time. Furthermore, until LP X has received an input event from Y, it cannot proceed (i.e., allow time to advance) because X does not necessarily ‘‘know’’ to an event will not arrive from Y. Since event combinations must be presented to LP X in time stamp order, the simulation is deadlocked.

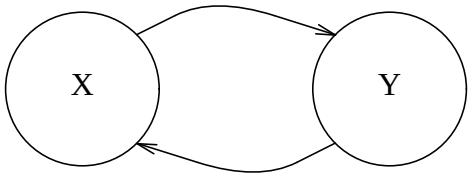

Figure. 8.1.

In the following sections we introduce three different ways to avoid deadlock in the Yaddes implementation of conservative synchronization. The Yaddes implementation currently does not support deadlock detection and recovery mechanisms. Hence, the simulation programmer must use one of the following methods to avoid deadlock.

8.1. Using `$output` and `$nulloutput`

One way to avoid deadlock is to emit events that do not correspond to events in the real-world system. Such null events carry no information, but are merely there to avoid deadlock. For example, in Figure. 8.1, LP X could send a null event to Y indicating that no output is present at time t. Process Y then responds with a null event back to X. Process X may then proceed since it now knows that no event will arrive from Y.

In the Yaddes system, null events can be emitted using the `$nulloutput` routine. The arguments to this routine are identical to those of the `$output` routine. The `$nulloutput` routine merely identifies the events in a simulation that are to avoid deadlock. Since the other synchronization methods do not suffer from deadlock, the `$nulloutput` function has no effect when using the other synchronization methods.

Note that events produced using `$nulloutput` and `$output` are indisinguishable at the destination (i.e., the receiver) LP. It is up to the simulation user to detect and ignore null events. This can be accomplished by reserving a certain event number for null events and explictly

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

ignoring such events.

8.2. Using `$ignore`

A second way to avoid deadlock is to have an LP announce that it is not expecting to receive events on a given input. For example, in Figure. 8.1, it may be that LP X knows that it will not receive any events from LP Y until after some time t. The `$ignore` function can be used to inform the synchronization method that no events will arrive from Y until after time t. Since the other synchronization methods do not suffer from deadlock, the `$ignore` function has no effect when using the other synchronization methods.

8.3. Using `$activate` and `$deactivate`

The third way to avoid deadlock is a variant of the previous method. Again, an LP announces that it is not expecting to receive events on a given input. The `$deactivate` function is used to inform the synchronization method that, until further notice, no input events will arrive on a given input. The `$activate` function is used to turn an input back on after it has been deactivated. Whereas in the case of the `$ignore` function, the LP must know a future time after which events may arrive, in the case of `$activate` and `$deactivate` the LP explicitly reactivates the input at some future time. Both of these approaches have their disadvantages: The `$ignore` routine requires an LP to predict the future whereas the `$deactivate` routine requires an LP to remember that an input has been deactivated.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Chapter 9.
Example: Queue and Server

This example illustrates the use of Yaddes to construct a simple queueing system simulation. This example illustrates the use of the Yaddes pseudo-random number generator package and the statistics gathering and reporting package. This example also illustrates the use of Yaddes features to avoid deadlock when using conservative distributed simulation.

The queueing system being simulated in this example is shown in Figure. 9.1. There are three physical processes in this system — the arrival process, the queue, and the server process. The Yaddes description of each of these physical processes is given in the following sections.

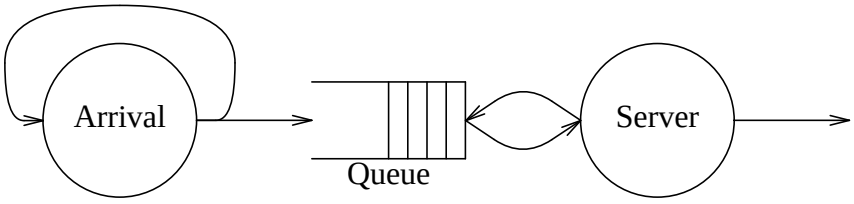

Figure. 9.1.

9.1. The Arrival LP

The Yaddes specification of the arrival LP is given below. A random arrival process is specified, i.e., the distribution of interarrival times is a biased exponential. The arrival LP has a single input and a single output. The arrival LP is a self-timed source of events, i.e., its output is connected back to its input.

The initial action of the arrival LP initializes the random number generator and a statistic that will keep track of the interarrival time. The initial action then selects a random interarrival time from a biased exponential distribution and outputs an event with that time less `EPSILON`. This event represents the arrival of a customer.

The quantity `EPSILON` is required in Yaddes since LPs are not allowed to take actions with zero delay. `EPSILON` represents the smallest resolvable non-zero time stamp increment.

Whenever an event arrives at the input to the arrival LP, it selects a new random interarrival time from a biased exponential distribution and schedules an event with that time. Since the output of the arrival LP is connected back to itself, it is a self-timed source of events.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
model Arrival
    inputs in
    outputs out
    state
    {
        STATISTIC_TYPE arrivalTime;
        RNG_TYPE producerRNG;
    }
    initial state { STATISTIC_INITIAL_VALUE, RNG_INITIAL_VALUE }
    action initial and not (in or final)
    {
        int arrTim;

        InitializeRNG ($pid + 1, &($state->producerRNG));
        DeclareStatistic (&($state->arrivalTime), POPULATION);
        arrTim = ExpDistRandVar (MEAN_ARRIVAL_TIME - BIAS_ARRIVAL_TIM
            &($state->producerRNG)) + BIAS_ARRIVAL_TIME;
        AccumulateStatistic (&($state->arrivalTime), arrTim, $time);
        $output ($out, $time + arrTim - EPSILON, 0);
        $settimelimit (10000);
    }
    action in and not (initial or final)
    {
        int arrTim;

        arrTim = ExpDistRandVar (MEAN_ARRIVAL_TIME - BIAS_ARRIVAL_TIM
            &($state->producerRNG)) + BIAS_ARRIVAL_TIME;
        AccumulateStatistic (&($state->arrivalTime), arrTim, $time);
        $output ($out, $time + arrTim, 0);
    }
    action final and not (initial or in)
    {
        ReportStatistic (&($state->arrivalTime),
            "inter arrival time");
    }
end model
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

9.2. The Queue LP

The Yaddes specification of the queue LP is given below. The queue LP has two inputs and one output. Input events from the arrival LP represent the arrival of a customer. Input events from the server LP represent a request for the next customer in the queue.

The initial action of the queue LP initializes two statistics: one to keep track of the (time-averaged) queue length, and one to keep track of the time customers spent in the queue. The initial action also deactivates the request input. This is needed to avoid deadlock. The protocol is that the server will only issue a request after it has finished servicing a customer. Since no customers have been sent to the server, no request will arrive. When the request input is deactivated, the queue LP sets a Boolean value in its state to indicate that a request is pending. This means that a customer can be forwarded directly to the server as soon as it arrives. Before forwarding the customer, the queue must reactivate the request input so that it can receive the next request from the server.

In addition to the initial and final actions, there are three possible event combinations that the queue LP must deal with: 1. the arrival of a customer, 2. the arrival of a request, and 3. the simultaneous arrival of an event and a customer. Generally, when a customer arrives, the queue LP enqueues the arrival time of that customer. When a request arrives, the first arrival time is dequeued. At each event, the queue LP updates the appropriate statistics.

```
model Queue
    inputs in, request
    outputs out
    state
    {
        STATISTIC_TYPE queueLength;
        STATISTIC_TYPE timeInQueue;
        int requestPending;
        int head;
        int tail;
        int count;
        int queue [MAX_QUEUE_LENGTH];
        int queueTime [MAX_QUEUE_LENGTH];
    }
    initial state
    {
        STATISTIC_INITIAL_VALUE,
        STATISTIC_INITIAL_VALUE,
        1,
        0,
        0,
        0
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
}

action initial and not (in or request or final)
{
    DeclareStatistic (&($state->queueLength), TIME_AVERAGED);
    DeclareStatistic (&($state->timeInQueue), POPULATION);
    $deactivate ($request);
    $settimelimit (10000);
}
action in and not (initial or request or final)
{
    if ($state->requestPending)
    {
        $state->requestPending = 0;
        $activate ($request);
        $output ($out, $time + EPSILON, $event [$in]);
    }
    else
    {
        if ($state->tail == MAX_QUEUE_LENGTH - 1)
            $state->tail = 0;
        else
            $state->tail += 1;
        if ($state->tail == $state->head)
        {
            (void) fprintf (stderr,
                "%s: queue overflow - client discarded\n", $name)
        }
        else
        {
            $state->queue [$state->tail] = $event [$in];
            $state->queueTime [$state->tail] = $time;
            AccumulateStatistic (&($state->queueLength),
                $state->count,
                $time + EPSILON);
            $state->count += 1;
        }
    }
}
action request and not (initial or in or final)
{
    int waitTime;
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
    if ($state->head == $state->tail)
    {
        $deactivate ($request);
        $state->requestPending = 1;
    }
    else
    {
        if ($state->head == MAX_QUEUE_LENGTH - 1)
            $state->head = 0;
        else
            $state->head += 1;
        $output ($out, $time + EPSILON,
            $state->queue [$state->head]);
        waitTime = $time - $state->queueTime [$state->head];
        AccumulateStatistic (&($state->timeInQueue), waitTime,
            0);
    AccumulateStatistic (&($state->queueLength),
    $state->count,
    $time + EPSILON);
        $state->count -= 1;
    }
}
action in and request and not (initial or final)
{
    int waitTime;

    if ($state->head == $state->tail)
    {
        $output ($out, $time + EPSILON, $event [$in]);
    }
    else
    {
        if ($state->head == MAX_QUEUE_LENGTH - 1)
            $state->head = 0;
        else
            $state->head += 1;
        $output ($out, $time + EPSILON,
            $state->queue [$state->head]);
        waitTime = $time - $state->queueTime [$state->head];
        AccumulateStatistic (&($state->timeInQueue), waitTime,
            0);
        if ($state->tail == MAX_QUEUE_LENGTH - 1)
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
                $state->tail = 0;
            else
                $state->tail += 1;
            $state->queue [$state->tail] = $event [$in];
            $state->queueTime [$state->tail] = $time;
        }
    }
    action final and not (initial or in or request)
    {
        AccumulateStatistic (&($state->queueLength), $state->count,
            $time + EPSILON);
        ReportStatistic (&($state->queueLength), "queue length");
        ReportStatistic (&($state->timeInQueue), "time in queue");
        (void) printf (
            "Queue: %d clients left in queue at end of simulation\n",
            $state->count);
    }
end model
```

9.3. The Server LP

The Yaddes specification of the server LP is shown below. The server LP has one input and two outputs. The input to the server LP represents the arrival of customers. The outputs of the server LP represent the departure of customers and the issuing of requests for new customers.

The initial action of the server initializes the pseudo-random number generator and a statistic that will keep track of the service times.

Whenever a customer arrives at the server, a service time is selected from a biased exponential distribution. Then, an output event is scheduled representing the departure of the customer and a request for a new customer is sent to the queue LP.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
model Server
    inputs in
    outputs request, out
    state
    {
        STATISTIC_TYPE serviceTime;
        RNG_TYPE serverRNG;
    }
    initial state { STATISTIC_INITIAL_VALUE, RNG_INITIAL_VALUE }
    action initial and not (in or final)
    {
        InitializeRNG ($pid + 1, &($state->serverRNG));
        DeclareStatistic (&($state->serviceTime), POPULATION);
        $settimelimit (10000);
    }
    action in and not (initial or final)
    {
        int servTim;

        servTim = ExpDistRandVar (MEAN_SERVICE_TIME - BIAS_SERVICE_TI
            &($state->serverRNG)) + BIAS_SERVICE_TIME;
        $output ($out, $time + servTim, $event [$in]);
        $output ($request, $time + servTim - EPSILON, 0);
        AccumulateStatistic (&($state->serviceTime), servTim,
            $time);
    }
    action final and not (initial or in)
    {
        ReportStatistic (&($state->serviceTime), "service time");
    }
end model
```

9.4. The complete specification

In order to complete the Yaddes specification, the LPs and connections must be declared. (In addition, some C housekeeping must be done.) The complete specification of the Queue and Server example is shown below.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
preamble {
#include <stdio.h>
#include <math.h>

#define MAX_QUEUE_LENGTH 32

#define MEAN_ARRIVAL_TIME 200
#define BIAS_ARRIVAL_TIME 2
#define MEAN_SERVICE_TIME 200
#define BIAS_SERVICE_TIME 2
#define EPSILON 1

#define MAX_LONG 0x7fffffff

extern int ExpDistRandVar ();

}
```

The arrival model specification goes here.

The queue model specification goes here.

The server model specification goes here.

```
process arrival : Arrival
process queue : Queue
process server : Server

connect arrival .out to arrival .in, queue .in
connect queue .out to server .in
connect server .request to queue .request

postamble {

    int ExpDistRandVar (mean, state)
    int mean;
    RNG_TYPE *state;
    {
        long randLong;
        int result;

        do
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
        {
            do
            {
                randLong = Random (state);
            }
            while (randLong == 0);
            result = (int) -((double) mean *
                log ((double) randLong /(double) MAX_LONG));
        }
        while (result == 0);
        return (result);
    }
}
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix A.
Distributed Simulation on a Transputer Network

This appendix describes how to compile and run Yaddes simulations on the Transputer network. The current implementation supports execution on an array ofl eight T414 transputers configured in a cube topology.

A.1. Preparing the Yaddes Source File

The Yaddes source file must be modified to eliminate any references to the standard I/O libraries. This is because the transputers on which the simulation runs do not have any attached I/O devices. Perform the following steps:

1. Delete any lines of the form `#include <stdio.h>` from the Yaddes source.
2. If you use file I/O (fopen, fclose, fprintf), insert a line of the following form

   ```
   #include io.h
   ```

3. Replace all occurrences of `printf (...)` by `fprintf (stdout, ...)`.
4. Use `yaddes2` to translate the Yaddes source file to a C program.
5. Edit the C program produced `yaddes2` and change the procedure name `main` to `Yaddes`.

A.2. Compiling the C program file

The C program produced by `yaddes2` must now be compiled for the T414 Transputer. Perform the following steps:

1. Compile the C program source using the 3L Parallel C compiler. Assuming the program is called `foo.c`, a command line of the following form should be used: `t4c foo`. The output of the compiler has the suffix `.bin`.

A.3. Linking the program

The `.bin` file must now be linked to the appropriate libraries. Perform the following steps:

1. Link the program to the appropriate kernel library and the stand-alone C run-time library. The kernel libraries have names of the form `~yaddes/yaddes/transp/lib/dist_xx.lib`, where the `xx` is either `el`, `ml`, `cm`, or `vt`. Assuming the program is called `foo.c`, a command line of the following form should be used: `ilink taskharn.t4x foo.bin ~yaddes/yaddes/transp/lib/dist_el.lib sacrtl.lib -o foo.c4x`. The output of this command is a file with suffix `.c4x`.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

2. Prepare the linked file for input to the configuration program. The purpose of this is to create a file with the appropriate format for downloading to the transputers. A command of the following form should be used: `iboot foo.c4x -c -o foo.d4x`. The output of this command is a file with suffix `.d4x`.

A.4. Configuring the file for execution

The program must now be configured for execution on a network of transputers. The configuration assigns tasks to processors and configures the software for the given hardware connection topology. Perform the following steps:

1. Configure the file for execution using the `configc` program. (Note, the 3L Parallel C utility called `config` has been renamed `configc` since there is a utility with the same name that comes with the Occam compiler.) A configuration file is available in `~yaddes/yaddes/tranps/lib/yaddes.cfg`. This file assumes that the program is called `slave.b4`. Change the name of your program appropriately using a command of the form: `mv foo.d4x slave.b4`
2. Now, invoke the configuration utility using a command of the form: `configc ~yaddes/yaddes/transp/lib/yaddes.cfg foo.b4`. The output of this command is a file with suffix `.b4`.

A.5. Configuring the Transputer Network

The crossbar switches in the Transputer network must be configured to establish the desired connection topology. This configuration only needs to be done once when the Transputers are turned on. To configure the Transputer network, perform the following steps:

1. Use the Module Motherboard Software package to set the configuration. Invoke the package with a command of the form: `mms2 ~yaddes/yaddes/transp/lib/softwire ~yaddes/yaddes/transp/lib/hardwire`.
2. The module motherboard software is menu driven. Use the ’r’ option to reset the Transputer network. Then use the ’i’ option to initialize the C004s. Then use the ’s’ option to set the programmable connections. The ’n’ option may now be used to verify the network topology.
3. Exit from the module motherboard software package using the ’q’ option.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

A.6. Running the Simulation

The `.b4` file produced above can booted and run on the network of transputers. Use the `iserver` utility to download and boot the program. A command of the following form is needed: `iserver -sb foo.b4`. Note, Yaddes option flags may be appended to the command as follows: `iserver -sb foo.b4 -w +s`.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix B.
Distributed Simulation on an Apollo Network

This appendix describes how to execute Yaddes simulations on a network of Apollo workstations.

B.1. Creating the Executable File

Creating a Yaddes distributed simulation to run on multiple Apollo nodes is similar to creating a Yaddes simulation to run on a single Apollo node. The only difference is in the libraries used.

1. Translate the Yaddes source file into a C program using `yaddes2`. Assuming a Yaddes source file `foo.y`, the command `yaddes2 foo.y` performs the translation to `foo.c`.
2. Compile the C program and link it to one of the Apollo distributed simulation kernels using the C compiler. The distributed Apollo kernels are dist-multi-list.a, dist-chandy-misra.a and dist-virtual-time.a. For example, using `foo.c` produced by yaddes2, the executable file, `foo`, can be created by the command `cc -o foo foo.c dist-multi-list.a` using the distributed multi-list kernel.

B.2. Running the Distributed Simulation

A copy of the simulation program must be initiated on each node and then the full simulation started.

1. For each Apollo node on which the distributed simulation will be run, start a copy of the simulation and indicate the processor number of that copy. Assuming the simulation program is called `foo`, the command `foo 1 [flags]` starts a copy on a node and assigns it a processor number of 1. The optional flags (see section 5.7.2) can be set at this stage or during execution using the `flagman` command.
2. Once all processors have a copy of the simulation, use the `dispatch` command to start the simulation. Assuming the simulation program is called `foo`, the command `dispatch foo` starts the simulation.

The multiple event-list kernel requires one extra processor to act as the scheduler. When using this kernel, the user must start n+1 processes instead of the usual n processes used in optimistic and conservative synchronization methods. Note, processes are numbered starting from 0.

After the simulation has been started (using `dispatch`) the `flagman` command can be used to change the print/trace options that may have been set on the command line. Executing `flagman` with the name of the simulation, as in `flagman foo`, allows processes executing the `foo` simulation to individually have their flags changed. See the `flagman` man page for

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

more information.

B.3. Simulation Cleanup

In the event of a distributed simulation terminating abnormally, it is necessary to remove superfluous Yaddes simulation entries from the Apollo NCS global location broker. This is done by the command `mrclean`. This program lists and prompts for entries in the NCS global location broker to be removed. For further information, see the `mrclean` man page.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix C.
Yaddes BNF

specification ::= *declarationList*

declarationList ::= ε | *declarationList declaration*

declaration ::= *constantDeclaration* | *modelDeclaration* | *processDeclaration* | *connectionDeclaration* | *procedureDeclaration* | *preamble* | *postamble* | *importStatement*

constantDeclaration ::= `constant` *identifier* = *manifestExpression*

modelDeclaration ::= `model` *identifier modelAttributeList* `end model` | `macro model` *identifier macroDeclarationList* `end model`

processDeclaration ::= `process` *variable onPart* `:` *identifier* | `process` *variable onPart* `:` *identifier* `{` *initializer-list* `}`

connectionDeclaration ::= `connect` *port* `to` *portList*

procedureDeclaration ::= `procedure` *identifier* `: initial` | `procedure` *identifier* `: final`

preamble ::= `preamble` `{` *program* `}`

postamble ::= `postamble` `{` *program* `}`

importStatement ::= `import` *symbolList* `from` *string-constant*

manifestExpression ::= *manifestTerm*

manifestTerm ::= *manifestProduct* | *manifestTerm* + *manifestProduct* | *manifestTerm* - *manifestProduct*

manifestProduct ::= *manifestFactor* | *manifestProduct* * *manifestFactor* | *manifestProduct* / *manifestFactor* | *manifestProduct* % *manifestFactor*

manifestFactor ::= *manifestItem* | - *manifestItem*

manifestItem ::= *integer-constant* | *manifestVariable* | (*manifestExpression*)

manifestVariable ::= *identifier* | *identifier* [*manifestExpression*]

modelAttributeList ::= ε | *modelAttributeList modelAttribute*

modelAttribute ::= *action* | *inputPart* | *outputPart* | *statePart* | *initialStatePart* | *constantDeclaration*

macroDeclarationList ::= ε | *macroDeclarationList macroDeclaration*

macroDeclaration ::= *constantDeclaration* | *processDeclaration* | *connectionDeclaration*

variable ::= *identifier* | *identifier* [*manifestExpression*] |

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

identifier `[` *manifestExpression* `..` *manifestExpression* `]`

onPart ::= ε | `on` *manifestExpression* |
`on` *manifestExpression* `..` *manifestExpression*

port ::= *portVariable* `.` *identifier* |
portVariable `.` *identifier* `[` *manifestExpression* `]` |
portVariable `.` *identifier* `[` *manifestExpression* `..` *manifestExpression* `]`

portList ::= *port* | *portList* `,` *port*

symbolList ::= *identifier* | *symbolList* `,` *identifier*

action ::= `action default {` *statement-list* `}` |
`action` *actionExpression* `{` *statement-list* `}`

inputPart ::= `inputs none` | `inputs` *variableList*

outputPart ::= `outputs none` | `outputs` *variableList*

statePart ::= `state {` *struct-decl-list* `}`

initialStatePart ::= `initial state {` *initializer-list* `}`

portVariable ::= *identifier* | *identifier* `[` *manifestExpression* `]` |
identifier `[` *manifestExpression* `..` *manifestExpression* `]`

actionExpression ::= *actionTerm*

variableList ::= *variable* | *variableList* `,` *variable*

actionTerm ::= *actionProduct* | *actionTerm* `or` *actionProduct*

actionProduct ::= *actionFactor* | *actionProduct* `and` *actionFactor*

actionFactor ::= *actionItem* | `not` *actionItem*

actionItem ::= `initial` | `final` | `nothing` | `something` | `anything` |
actionVariable | `(` *actionExpression* `)`

actionVariable ::= *identifier* | *identifier* `[` *manifestExpression* `]` |
identifier `[` *manifestExpression* `or ..` *manifestExpression* `]` |
identifier `[` *manifestExpression* `and ..` *manifestExpression* `]`

identifier ::= *see Kernighan & Ritchie, p. 179.*

integer-constant ::= *see Kernighan & Ritchie, p. 180.*

string-constant ::= *see Kernighan & Ritchie, p. 180.*

struct-decl-list ::= *see Kernighan & Ritchie, p. 216.*

initializer-list ::= *see Kernighan & Ritchie, p. 217.*

statement-list ::= *see Kernighan & Ritchie, p. 217.*

program ::= *see Kernighan & Ritchie, p. 218.*

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix D.
Kernel Statistics

D.1. Libraries for Sequential Simulation

(event-list.a, event_li.lib and dist_el.lib)

number of events posted — The number of events inserted into the event list. Events are inserted into the event list as a result of calls to the `$output` routine.

number of events not posted — The number of events not inserted into the event list. Events are not inserted into the event list when the `$nulloutput` routine is called.

number of events processed — The number of events removed from the event list and processed. This number may be less than the number of events inserted into the event list as a result of the simulation reaching the time limit set using a call to the `$settimelimit` routine.

number of generations — A new generation is said to begin every time the simulation clock advances. The number of generations is simply the number of different values of the clock during the simulation.

average generation size — The size of a generation is defined as the number of model calls made in that generation. The average generation size is the average number of model calls over all generations.

number of model calls — The total number of model calls made during the simulation.

average model calls per generations — Exactly the same as the average generation size above.

number of predictions — The number of events scheduled. Events are scheduled by the `$output` routine.

average prediction — The average difference between the current simulation time of a process scheduling an event and the time at which the event is to occur.

D.2. Libraries for Distributed Simulation using Multiple Event Lists

(multi-list[.vax].a, multi_li.lib and dist_ml.lib)

number of messages sent — The total number of messages sent. This includes event messages and control messages. Event messages are sent as a result of calls to the `$output` routine. Control messages are used to synchronize the multiple event lists.

number of messages received — The total number of messages received. This should be exactly the same as the total number of messages sent.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

number of null messages not sent — The total number of event messages not sent. Event messages are not sent when the `$nulloutput` routine is called. (These messages are not included in the total number of messages sent.)

number of control messages — The total number of control messages sent. Control messages are used to synchronize the multiple event lists.

number of events posted — The number of events inserted into local event lists. Events are inserted into local event lists as a result of calls to the `$output` routine.

number of events not posted — The number of events not inserted into local event lists. Events are not inserted into local event lists when the `$nulloutput` routine is called.

number of events processed — The number of events removed from the local event lists and processed. This number may be less than the number of events inserted into the local event lists as a result of the simulation reaching the time limit set using a call to the `$settimelimit` routine.

number of context switches — *[.vax version only]* The number of context switches used in simulating the multiprocessor.

number of generations — A new generation is said to begin every time the simulation clock advances. The number of generations is simply the number of different values of the clock during the simulation.

number of model calls — The total number of model calls made during the simulation.

number of model calls per generation — The average number of model calls made in each generation.

number of envelopes — The number of processors in the simulated multiprocessor.

average busy fraction — The average fraction of processors busy in each generation. A processor is busy if it has at least one model call to make. This figure does not take account of the fact that some processors may have more than one model call to make a generation.

number of predictions — *[portable and T800 versions only]* The number of events scheduled. Events are scheduled by the routine.

average prediction — *[portable and T800 versions only]* The average difference between the current simulation time of a process scheduling an event and the time at which the event is to occur.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

D.3. Libraries for Conservative Simulation

(chandy-misra[.vax].a, chandy_m.lib and dist_cm.lib)

number of messages sent — The total number of messages sent. This includes event messages and null messages and a message sent by the kernel to each process when deadlock is detected (e.g., at the end of the simulation). Event messages are sent as a result of calls to the `$output` routine. Null messages are sent as a result of calls to the `$nulloutput` routine.

number of messages received — The total number of messages received. This should be exactly the same as the total number of messages sent.

number of null messages — The total number of null messages sent. Null messages are sent as a result of calls to the `$nulloutput` routine.

number of context switches — *[.vax version only]* The number of context switches used in simulating the multiprocessor.

number of generations — *[.vax version only]* The basic cycle executed by the envelope involves receiving a message, possibly making some model calls (which may send messages), and then receiving the next message. In each generation, all envelopes are allowed to receive at most one message.

average generation size — *[.vax version only]* The size of a generation is the number of envelopes that process a message in that generation. The average generation size is the average over all generations of the the size of a generation.

number of model calls — The total number of model calls made during the simulation.

average model calls per generation — *[.vax version only]* The average number of models calls in each generation.

number of predictions — *[portable and T800 versions only]* The number of events scheduled. Events are scheduled by the `$output` routine.

average prediction — *[portable and T800 versions only]* The average difference between the current simulation time of a process scheduling an event and the time at which the event is to occur.

D.4. Libraries for Optimistic Simulation

(virtual-time[.vax].a, virtual_.lib and dist_vt.lib)

number of messages sent — The total number of messages sent. This includes event messages, antimessages, and token messages. Event messages are sent as a result of calls to the `$output` routine. Antimessages are sent as a result of rolling back states. Token messages are used to implement fossil collection.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

number of messages received — The total number of messages received. This should be exactly the same as the total number of messages sent.

number of null messages not sent — The total number of event messages not sent. Event messages are not sent when the `$nulloutput` routine is called. (These messages are not included in the total number of messages sent.)

number of antimessages — The total number of antimessages sent. Antimessages are sent as a result of rolling back states.

number of token messages — The total number of token messages. A circulating token is used to implement fossil collection.

number of context switches — *[.vax version only]* The number of context switches used in simulating the multiprocessor.

number of model calls — The total number of model calls made during the simulation.

number of states rolled back — The total number of states rolled back. This is the number of states that were erroneously entered and had to be discarded during roll back.

number of predictions — *[portable and T800 versions only]* The number of events scheduled. Events are scheduled by the `$output` routine.

average prediction — *[portable and T800 versions only]* The average difference between the current simulation time of a process scheduling an event and the time at which the event is to occur.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix E.
Yaddes Compiler Output

In this appendix, the output of the Yaddes compiler will be described. The following Yaddes input file uses many of the yaddes features. The output produced by the Yaddes compiler for this input file is shown following the input file.

Yaddes Source:

```
model FooBar
    inputs Alpha, Beta
    outputs Xi, Delta
    state { int Epsilon; }
    initial state { 1234 }
    action initial and not (Alpha or Beta or final)
    { $settimelimit (10000); }
    action Alpha and not (initial or Beta or final)
    { $deactivate ($Beta); }
    action (Beta or (Alpha and Beta)) and not (initial or final)
    {
        if ($eventoccurred ($Alpha))
            $output ($Xi, $time + 10, $event [$Alpha]); }
    action nothing { $ignore ($Alpha, $time + 20); }
    action final and not (initial or Alpha or Beta)
    { }
end model

process Zero on 0 : FooBar
process One on 1 : FooBar

connect Zero .Xi to One .Alpha
connect Zero .Delta to One .Beta
connect One .Xi to Zero .Alpha
connect One .Delta to Zero .Beta

procedure InitialProcedure : initial
procedure FinalProcedure : final
```

Yaddes Compiler Output:

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
 1  int xxNumber[]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
 2  16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31};
 3  #ifndef YADDES_DOT_H
 4  #include "yaddes.h"
 5  #endif
 6  typedef struct {
 7  #line 4 "foo.y"
 8   int Epsilon;
 9  } xxFooBarSTATE;
10  char *xxFooBarInputs [] = {
11  "Alpha",
12  "Beta",
13  0};
14  char *xxFooBarOutputs [] = {
15  "Xi",
16  "Delta",
17  0};
18  void FooBar (xxState, xxTime, xxMask, xxEventList,
19  xxInputList, xxOutputList,
20  xxInputTable, xxOutputTable,
21  xxChannelTime, xxQueueHead,
22  xxTimeLimit, xxActiveChannels)
23  xxFooBarSTATE *xxState;
24  int xxTime;
25  int xxMask;
26  EVENT *xxEventList;
27  char **xxInputList;
28  char **xxOutputList;
29  int *xxInputTable;
30  int *xxOutputTable;
31  int *xxChannelTime;
32  MESSAGE_QUEUE *xxQueueHead;
33  int *xxTimeLimit;
34  int *xxActiveChannels;
35  { switch (xxMask) {
36  case 4:
37  {
38  #line 7 "foo.y"
39   SET_TIME_LIMIT (10000);
40  };
41  break;
42  case 1:
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
43  {
44  #line 9 "foo.y"
45   DEACTIVATE (1);
46  };
47  break;
48  case 2:
49  case 3:
50  {
51  #line 11 "foo.y"
52
53          if (EVENT_OCCURRED (0))
54              OUTPUT (0, xxTime + 10, xxEventList [0]);
55  };
56  break;
57  case 0:
58  {
59  #line 14 "foo.y"
60   IGNORE (0, xxTime + 20);
61  };
62  break;
63  case 8:
64  {
65  #line 16 "foo.y"
66
67  };
68  break;
69  default:
70  DefaultAction (xxMask, xxEventList, xxInputList);
71  break;
72  }}
73  xxFooBarSTATE xxFooBarStateTable [] = {
74  {
75  #line 5 "foo.y"
76   1234
77  },
78  {
79  #line 5 "foo.y"
80   1234
81  },
82  {0}};
83  int xxInputOutputTables [] = {
84  2,6,1,1,1,0,1,1,1,1,
```

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

```
 85  2,6,1,1,1,0,1,1,1,1,
 86  2,6,1,0,0,0,1,0,0,1,
 87  2,6,1,0,0,0,1,0,0,1,
 88  0};
 89  main (argc, argv)
 90  int argc; char **argv;
 91  {
 92  (void) InterpretArguments (argc, argv);
 93  (void) InitialProcedure (argc, argv);
 94  (void) LaunchProcess (Envelope,
 95  (int) (char *) FooBar,
 96  (int) &(xxFooBarStateTable [0]),
 97  sizeof (xxFooBarSTATE),
 98  (int) xxFooBarInputs,
 99  (int) xxFooBarOutputs,
100  (int) &(xxInputOutputTables [0]),
101  (int) &(xxInputOutputTables [10]),
102  0,
103  "Zero",2048);
104  (void) LaunchProcess (Envelope,
105  (int) (char *) FooBar,
106  (int) &(xxFooBarStateTable [1]),
107  sizeof (xxFooBarSTATE),
108  (int) xxFooBarInputs,
109  (int) xxFooBarOutputs,
110  (int) &(xxInputOutputTables [20]),
111  (int) &(xxInputOutputTables [30]),
112  1,
113  "One",2048);
114  DispatchProcesses ();
115  (void) FinalProcedure ();
116  return (0);
117  }
```

Every model specification produces a procedure with the same name. In the example, the `FooBar` model produces the `FooBar` procedure (see line 18).

The input specification of a model produces a declaration of an array of strings. The name of the variable is `xx` followed by the model name followed by `Inputs`. In the example, the `FooBar` model's input specification produces the `xxFooBarInputs` variable declaration (see line 10). The array is initialized with strings containing the input names.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

The output specification of a model produces a declaration of an array of strings. The name of the variable is `xx` followed by the model name followed by `Outputs`. In the example, the `FooBar` model's output specification produces the `xxFooBarOutputs` variable declaration (see line 14). The array is initialized with strings containing the output names.

The state specification of a model produces a structure type definition. The name of the type is `xx` followed by the model name followed by `STATE`. In the example, the `FooBar` model's state specification produces the `xxFooBarSTATE` type definition (see lines 6-9).

The initial state specification produces a structure initializer value. Its use is described below. In the example, the `FooBar` model's initial state specification produces the values used on lines 76 and 80.

The body of the procedure generated from a model consists of a single switch (case) statement. Each action specification produces one of the cases in the case statement. The case selector is a bit mask in which the i^{th} bit is set if the i^{th} input is specified in the action selector. The number of an input is its index in the `xx...Inputs` array. For the initial action the n^{th} bit is set and for the final action the $(n+1)^{th}$ bit is set where n is the number of inputs. In the example where there are two inputs, `action initial` translates to `case 4` (2^2), `action final` translates to `case 8` (2^3), `action nothing` translates to `case 0`, `action Beta` translates to `case 2` (2^1), `action Alpha` translates to `case 1` (2^0), and `action (Alpha and Beta)` translates to `case 3` (2^0+2^1).

Inside the procedure, references to the variables `$time`, `$event`, and `$state` are translated to references to the local variables `xxTime`, `xxEventList`, and `xxState` (see lines 54 and 60). References to input and output constants, `$Alpha`, `$Beta`, `$X1` and `$Delta`, are converted to their numeric values (see lines 45, 53, 54 and 60). References to input and output constant arrays are converted to expressions using the array xxNumber (line 1) to determine their value.

Each process specification produces a variable declaration. The declared variable is an element of an array. The name of the array is the name of the model of which the process is an instance preceded by `xx` and followed by `StateTable`. The type of the array is the type defined by the state specification of the specified model. In the example, the `process Zero` and `process One` specifications produce the declaration of the two element array called `xxFooBarStateTable` (see line 73). The first entry in the array corresponds to `processZero`, the second entry corresponds to `processOne`. The array is initialized to the value obtained from the initial state specification of the model. In the example, the `xxFooBarStateTable` array entries are initialized to `1234` (see lines 76 and 80).

Each process also has two integer sequences associated with it. These sequences are all stored in the common array called `xxInputOutputTables` (see line 83). In the example, the `process Zero` specification produces the entries on lines 84 and 85. The `process One` specification produces the entries on lines 86 and 87. These entries represent information derived from the connection specifications and the `xx...Inputs` and `xx...Outputs` declarations and the process `on` specifications. These arrays are interpreted as follows. The first

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

number is the number of inputs or outputs that the process has. Let this number be n. The first n numbers are indices into the array indicating where data about the i^{th} input or output $0 \leq i \leq n-1$ begins. Consider the entries on line 84 of the example. The first number in the array, `2` specifies that there are 2 inputs. Information about input number 0 begins at index 2 and information about input number 1 begins at index 6. (Note the double meaning of the first number.) Note that the name of input or output i can be obtained from the appropriate entry in `xx..Inputs` or `xx..Outputs` array entry. The information about an input or output consists of a number which specifies the fan-in or fan-out of the input. Let this number be f. Following this number are f triples. The last number in each the `xxInputOutputTables` is just noise and should be ignored. Consider line 84 of the example. Information about input number zero begins at index 2 in the list. The number there is `1`. This indicates that the fan-in is 1. Following this number, there is 1 triple, `1,1,0`. The triples consist of a processor number, a process number, and an output number. This says that there is a process on processor 1, the number of that process is 1, and its output number 0 is connected to this input. The output table on line 85 is constructed in a similar fashion.

A single main procedure is emitted. This main procedure calls `LaunchProcess` once for each process specified in the source. A number of arguments are passed to the `LaunchProcess` procedure. Of note is the 9th argument, which is the process number assigned to that process. In the example, process `Zero` is assigned process number `0` on line 102.

Simulation begins when the main process calls `DispatchProcesses`. Before simulation begins, the initial procedure is called. After simulation ends, the final procedure is called. In the example, the initial procedure is `InitialProcedure` (line 93) and the final procedure is `FinalProcedure` (line 115).

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix F.
Pitfalls

There are a number of pitfalls that a Yaddes programmer should be aware of when writing a Yaddes simulation.

1. Deadlock in Chandy-Misra — When using the Chandy-Misra simulation mechanism, it is possible to write a simulation which will deadlock. This occurs because a model will not be called until it can determine the time for the next event, or null event, on all inputs to the model. `$nulloutput`, input deactivation and ignoring inputs (see section 8) have to be used to prevent deadlock in these situations.
2. Zero delay — In Yaddes, it is not possible to schedule events with zero delay. This means that all events must be scheduled to occur at least one clock tick after the current simulation time of the process when it schedules the event. See the Queue and Server model (section 9) for an example of this problem.
3. Use of `#define`s — Yaddes is a preprocessor to the C language and so is used before the macro processor of the C compiler. This means that any values used in the yaddes or yaddes2 translation must not make use of C language macros (`#define`), file inclusion (`#include`), or conditional compilation (`#ifdef`). Places where values must be determined by Yaddes include model input and output declarations, action statements, process declarations and connection specification. C language preprocessor features can be used within the preamble and postamble code and in the code executed for the action statements and state declaration.
4. Action `nothing` — statements that are to execute in response to a no event action will only do so when the Chandy-Misra simulation mechanism is used since this action can only occur when inputs are ignored or deactivated. When using all other simulation mechanisms, `$ignore` and `$deactivate` commands are null operations.
5. Globals — Global variables can be used in the uniprocessor Yaddes simulations but these variables are not shared among multiple processors in distributed simulations.
6. Time 0 — The initial action occurs at time 0. The initial action will never be combined with other events. Therefore, events must never be scheduled with timestamp 0.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix G.
Yaddes `man` Pages

G.1. `dispatch`

NAME

`dispatch` — Start and time a distributed Yaddes simulation

SYNOPSIS

`dispatch` *[file]*

DESCRIPTION

`dispatch` is used to start and time a distributed Yaddes simulation. The system to be simulated is specified using the Yaddes specification language. This specification is translated to C code using `yaddes2`(p) which is then compiled using `cc`(1) and linked to an execution mechanism library using `ld`(1).

Three libraries, `dist-multi-list.a`, `dist-chandy-misra.a`, and `dist-virtual-time.a`, allow distributed simulation on Apollo workstations running DOMAIN/IX and NCS. When a simulation program is linked to one of these libraries, a simulation server is created. The simulation is run by invoking the desired number of servers (on one or more workstations) and using the `dispatch` program to start the simulation.

`dispatch` requires the name of the object file of the simulation program to be run as its one and only argument.

`dispatch` prints the elapsed real time required to complete the simulation.

SEE ALSO

`yaddes2`(p), `mrclean`(p), and `flagman`(p).

G.2. `flagman`

NAME

`flagman` — Manipulate the print/trace option flags of a running distributed Yaddes simulation

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

SYNOPSIS

`flagman` *[file]*

DESCRIPTION

`flagman` is used to change the print and trace flag settings of a running distributed Yaddes simulation. The flags of a running simulation server can be changed either before or after being dispatched (see `dispatch`(p)).

`flagman` requires the name of the object file of the simulation server that is currently running as its one and only argument.

`flagman` will list all servers running from the given object file. The user can interactively select a server, examine its current flag settings, and change the flag settings. For a list of server flags and their meanings, see `yaddes2`(p).

SEE ALSO

`yaddes2`(p), `dispatch`(p), and `mrclean`(p).

G.3. `mrclean`

NAME

`mrclean` — Clean up superfluous Yaddes simulation entries from the NCS global location broker

SYNOPSIS

`mrclean`

DESCRIPTION

`mrclean` is used to clean up the NCS global location broker in the event that it contains out-dated entries. Out-dated entries can occur if a Yaddes distributed simulation server is terminated prematurely and does not get to unregister itself.

`mrclean` requires no arguments.

`mrclean` will list all Yaddes simulation servers registered with the global location broker and inquire whether or not to unregister the server.

SEE ALSO

`yaddes2`(p), `dispatch`(p), and `flagman`(p).

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

G.4. yaddes

NAME

yaddes2 Yaddes simulation specification language compiler

SYNOPSIS

yaddes2 *[file[.y]]*

DESCRIPTION

yaddes2 is a compiler for the Yaddes simulation specification language. The input is a specification of the system to be simulated. The output is a file containing a C language program that can be compiled with the cc(1) compiler.

If yaddes2 is invoked with no arguments, it reads standard input and writes to standard output. If yaddes2 is invoked with an argument, it reads its input from the specified file name. f the input file name ends in .y, yaddes2 writes output into a file the name of which is obtained by replacing the .y with .c. If the input file name does not end in .y, yaddes2 writes output into a file the name of which is obtained by appending .c to the input file name.

The resulting C language program is compiled using cc(1) and then linked to an execution mechanism library using ld(1).

SIMULATION OPTIONS

+a Equivalent to specifying all of +i +k +m +o +r +p +s +t.

-h Print out a usage summary. The simulation is not run. +h Print out a usage summary. The simulation is not run. +i Trace calls to $ignore, $activate, and $deactivate. (Default: tracing disabled.) +k Trace kernel activity. (Default: tracing disabled.) +m Print detailed message traffic statistics. (Default: printing disabled.)

+o Trace calls to $output and $nulloutput (Default: tracing disabled.) +r Print out the library name and revision number. (Default: printing disabled.) +p Print a post-mortem. (Default: printing disabled.) +s Print message traffic statistics summary. (Default: printing disabled.) +t Trace model calls. (Default: tracing disabled.) -w Disable warnings. (Default: warnings enabled).

FILES

~yaddes/yaddes/lib/dist-chandy-misra.a	
file.y	input file
file.c	output file
~yaddes/yaddes/inc/yaddes.h	include file

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`~yaddes/yaddes/lib/event-list.a`	library (portable)
`~yaddes/yaddes/lib/multi-list.a`	library (portable)
`~yaddes/yaddes/lib/chandy-misra.a`	library (portable)
`~yaddes/yaddes/lib/virtual-time.a`	library (portable)
`~yaddes/yaddes/lib/multi-list.vax.a`	library (VAX only)
`~yaddes/yaddes/lib/chandy-misra.vax.a`	library (VAX only)
`~yaddes/yaddes/lib/virtual-time.vax.a`	library (VAX only)
`~yaddes/yaddes/lib/dist-multi-list.a`	library (Apollo DOMAIN/IX with NCS only)
`~yaddes/yaddes/lib/dist-chandy-misra.a`	library (Apollo DOMAIN/IX with NCS only)
`~yaddes/yaddes/lib/dist-virtual-time.a`	library (Apollo DOMAIN/IX with NCS only)

SEE ALSO

`dispatch`(p), `mrclean`(p), and `flagman`(p).

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix H.
Yaddes Compile-Time Error Messages

`action selector expression overlap`: Two different action selector expressions specify the same event combination.

`can't open ...`: The specified file does not exist or is access-protected.

`cannot generate code: symbol $... invalid type`: The specified symbol has an inappropriate type in this context.

`cannot generate code: symbol $... not declared`: The specified symbol has not been declared in this context.

`division by zero, replaced with zero`: While evaluating the expression at compile time, a divide-by-zero error occurred.

`file ... not found`: The specified file does not exist or is access-protected.

`import failure, invalid symbol ... ignored`: The specified symbol is of an inapproriate type for importing.

`import failure, symbol ... not found`: The specified symbol was not declared in the import file (nor in any other transitively imported file).

`invalid array size, symbol ... ignored`: The symbol used in the array-size declaration is of an inappropriate type or has a negative value.

`invalid processor number, replaced with 0`: The processor number expression has a negative value.

`invalid processor range, replaced with 0`: One (or both) of the processor range expressions are negative or the lower bound is greater than the upper bound.

`invalid range, symbol ... ignored`: The symbol used in the range declaration is of an inappropriate type.

`invalid subscript in action expression ignored`: The subscripted symbol is not an array.

`invalid subscript in connection statement ignored`: The subscripted symbol is not an array.

`invalid subscript in manifest expression ignored`: The subscripted symbol is not an array.

`invalid subscript range in action expression, replaced with false`: The range is either not contained in the declared range or the lower bound is greater than the upper bound.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`invalid subscript range in connection statement, ignored`: The range is either not contained in the declared range or the lower bound is greater than the upper bound.

`invalid subscripted symbol ... in manifest expression replaced with 0`: The symbol is of an inappropriate type in this context.

`invalid symbol ... ignored`: The symbol is not valid in this context.

`invalid symbol ... in action expression, replaced with false`: The symbol is of an inappropriate type in this context.

`invalid symbol ... in connection statement ignored`: The symbol is of an inappropriate type in this context.

`invalid symbol ... in manifest expression, replaced with 0`: The symbol is of an inappropriate type in this context.

`macro models not implemented yet`: Macro models have not yet been implemented.

`no code generated due to errors`: Errors have occurred been detected during parsing that prevent the generation of code (but that did not prevent the completion of parsing).

`process ... assigned to processor 0`: The specified process has been assigned to processor 0 by default.

`range size mismatch in connection statement`: The number of outputs specified in the connection statement does not match the number of inputs specified.

`redeclaration of symbol ... hides earlier one`: The specified symbol is already declared in this context. In this case, the redeclaration hides the earlier declaration.

`redeclaration of symbol ... ignored`: The specified symbol is already declared in this context. In this case, the redeclaration is ignored.

`subscript out of range, expression replaced with 0`: The subscript expression value is out of range. The value of the subscripted symbol (not just the subscript) is replaced with 0.

`subscript out of range, replaced with false`: The subscript expression value is out of range. The value of the subscripted symbol (not just the subscript) is replaced with false.

`symbol ... missing subscript in action expression, replaced with false`: The specified symbol is missing a subscript.

`symbol ... missing subscript in connection statement, ignored`: The specified symbol is missing a subscript.

`symbol ... missing subscript in manifest expression, replaced with 0`: The specified symbol is missing a subscript.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`too many errors`: Too many errors have been encountered during parsing. Parsing is terminated. `too many inputs, symbol ... ignored`: The model in which the specified symbol is declared has too many inputs.

`undefined symbol ... ignored`: The specified symbol has not been declared.

`undefined symbol ... in action expression, replaced with false`: The specified symbol has not been declared.

`undefined symbol ... in connection statement ignored`: The specified symbol has not been declared.

`undefined symbol ... in manifest expression replaced with zero`: The specified symbol has not been declared.

`undefined symbol ... in manifest expression, replaced with zero`: The specified symbol has not been declared.

`unexpected connection to ... input ignored`: The specified input of the given logical process already has something connected to it.

`unexpected default action ignored`: A default action statement is inappropriate in this context.

`unexpected final procedure statement ignored`: A final procedure declaration is inappropriate in this context.

`unexpected initial procedure statement ignored`: An initial procedure declaration is inappropriate in this context.

`unexpected initial state statement ignored`: An initial state declaration is inappropriate in this context.

`unexpected input statement ignored`: An input declaration is inappropirate in this context.

`unexpected output statement ignored`: An output declaration is inappropirate in this context.

`unexpected postamble statement ignored`: A postamble statement is inappropirate in this context.

`unexpected preamble statement ignored`: A preamble statement is inappropirate in this context.

`unexpected state statement ignored`: A state declaration is inappropirate in this context.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Appendix I.
Run-Time Error Messages

I.1. `chandy-misra.a`**:**

`AccumulateStatistic: uninitialized or corrupt statistic reset`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`AllocateMessage: out of memory`: The kernel is unable to allocate further message buffers due to memory exhaustion.

`DispatchProcesses: already done`: A second call has been made to `DispatchProcesses`. This should never happen.

`Envelope: got a message after final action done`: The envelope of a logical process that has terminated (i.e., executed its final action) received a message. This is usually the result of scheduling events with time-stamps after the simulation time limit. This is usually harmless.

`Envelope: non-existent envelope invoked`: An envelope without a corresponding logical process has been invoked. This should never happen.

`InitializeStatistic: unknown type, ... assumed`: The type argument in the call to `InitializeStatistic` is unknown.

`LaunchProcess: no more memory`: The kernel is unable to allocate memory for more logical processes due to memory exhaustion.

`LaunchProcess: too many processes`: Too many processes have been declared. Recompile the kernel using more process descriptors.

`Receive: should not get here`: This should never happen.

`Send: invalid message destination`: A message was sent with an invalid destination address. This should never happen.

`statistics uninitialized or corrupt`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`Warning: attempted send to non-existent process ignored`: A message was sent with an invalid destination address. This should never happen.

`Warning: process ... discarding input ... chan time ... msg time ... event ...`: A message has arrived on an input to a logical process that has been `$deactivated` or `$ignored`.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`Warning: process ... no action for: ...`: An event combination has occurred for which there is no action specified.

`Warning: process ... time ... input ... already activated`: An input that is already active has been `$activate`d.

`Warning: process ... time ... input ... already deactivated`: An input that is already inactive has been `$deactivate`d.

`Warning: RelinquishProcessor: has no effect`: This should never happen.

I.2. `event-list.a`**:**

`AccumulateStatistic: uninitialized or corrupt statistic reset`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`AllocateMessage: out of memory`: The kernel is unable to allocate further message buffers due to memory exhaustion.

`DispatchProcesses: time monotonicity violation`: An attempt has been made to schedule an event into the past. All events must have time stamps greater than the current simulation time.

`Envelope: should not be called`: This should never happen.

`InitializeStatistic: unknown type, ... assumed`: The type argument in the call to `InitializeStatistic` is unknown.

`LaunchProcess: too many processes`: Too many processes have been declared. Recompile the kernel using more process descriptors.

`Receive: should not be called`: This should never happen.

`Send: event collision`: An attempt has been made to schedule two simultaneous events on a single output arc. All events on each output from a logical process must have unique time stamps and must be output in increasing time order.

`Send: invalid message destination`: A message was sent with an invalid destination address. This should never happen.

`statistics uninitialized or corrupt`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`Warning: ACTIVATE has no effect`: The `$activate` routine has no effect when using this kernel.

`Warning: DEACTIVATE has no effect`: The `$deactivate` routine has no effect when using this kernel.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`Warning: IGNORE has no effect`: The `$ignore` routine has no effect when using this kernel.

`Warning: process ... no action for: ...`: An event combination has occurred for which there is no action specified.

`Warning: RelinquishProcessor: has no effect`: This should never happen.

`Warning: time limit cannot be increased`: A call to `$settimelimit` was made that attempted to increase the time limit. This is not possible.

I.3. `multi-list.a`**:**

`AccumulateStatistic: uninitialized or corrupt statistic reset`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`AllocateMessage: out of memory`: The kernel is unable to allocate further message buffers due to memory exhaustion.

`DeclareTask: too many tasks`: Too many tasks have been declared. Recompile the kernel using more task descriptors.

`DispatchProcesses: grave error`: This should never happen.

`DispatchProcesses: out of memory`: The kernel is unable to dispatch processes due to memory exhaustion.

`Envelope: got a message after final action done`: The envelope of a logical process that has terminated (i.e., executed its final action) received a message. This is usually the result of scheduling events with time-stamps after the simulation time limit. This is usually harmless.

`Envelope: non-existent envelope invoked`: An envelope without a corresponding logical process has been invoked. This should never happen.

`InitializeStatistic: unknown type, ... assumed`: The type argument in the call to `InitializeStatistic` is unknown.

`LaunchProcess: too many processes`: Too many processes have been declared. Recompile the kernel using more process descriptors.

`Post: invalid message destination`:

`PostEvent: event collision`: An attempt has been made to schedule two simultaneous events on a single output arc. All events on each output from a logical process must have unique time stamps and must be output in increasing time order.

`Receive: should not get here`: This should never happen.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`RelinquishProcessor: has no effect`: This should never happen.

`Send: invalid message destination`: A message was sent with an invalid destination address. This should never happen.

`StartTasks: already done`: A second call was made to `StartTasks`. This should never happen.

`statistics uninitialized or corrupt`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`task ... cannot increase time limit`: A call to `$settimelimit` was made that attempted to increase the time limit. This is not possible.

`task ... event collision`: An attempt has been made to schedule two simultaneous events on a single output arc. All events on each output from a logical process must have unique time stamps and must be output in increasing time order.

`task ... received bad completion message`: This should never happen.

`task ... received bad message`: This should never happen.

`task ... received deadlock message`: This should never happen.

`task ... time monotonicity violation`: An attempt has been made to schedule an event into the past. All events must have time stamps greater than the current simulation time.

`Warning: ACTIVATE has no effect`: The `$activate` routine has no effect when using this kernel.

`Warning: attempted send to non-existent task ignored`: A message was sent with an invalid destination address. This should never happen.

`Warning: IGNORE has no effect`: The `$ignore` routine has no effect when using this kernel.

I.4. `virtual-time.a`:

`AccumulateStatistic: uninitialized or corrupt statistic reset`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`AllocateMessage: out of memory`: The kernel is unable to allocate further message buffers due to memory exhaustion.

`AllocateState: out of memory`: The kernel is unable to allocate further state buffers due to memory exhaustion.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`DispatchProcesses: already done`: A second call has been made to `DispatchProcesses`. This should never happen.

`Envelope: garbage collection error`: An error has occurred during garbage collection. This should never happen.

`Envelope: got a message after final action done`: The envelope of a logical process that has terminated (i.e., executed its final action) received a message. This is usually the result of scheduling events with time-stamps after the simulation time limit. This is usually harmless.

`Envelope: non-existent envelope invoked`: An envelope without a corresponding logical process has been invoked. This should never happen.

`InitializeStatistic: unknown type, ... assumed`: The type argument in the call to `InitializeStatistic` is unknown.

`LaunchProcess: too many processes`: Too many processes have been declared. Recompile the kernel using more process descriptors.

`process ... annihilation target not found`: An anti-message was sent, but the corresponding message was not found. This should never happen.

`process ... cannot roll back`: A process is attempting to roll back to a point in time for which it has no checkpointed state information. This should never happen.

`process ... event collision`: An attempt has been made to schedule two simultaneous events on a single output arc. All events on each output from a logical process must have unique time stamps and must be output in increasing time order.

`process ... no previous state`: A process is attempting to roll back to a point in time for which it has no checkpointed state information. This should never happen.

`process ... received unknown message type`: This should never happen.

`process ... should never get here`: This should never happen.

`Receive: should not get here`: This should never happen.

`Send: invalid message destination`: A message was sent with an invalid destination address. This should never happen.

`statistics uninitialized or corrupt`: The state of a statistic is invalid. Either the state has not been properly initialized or it has been corrupted during the course of the simulation.

`Warning: ACTIVATE has no effect`: The `$activate` routine has no effect when using this kernel.

`Warning: attempted send to non-existent process ignored`: A message was sent with an invalid destination address. This should never happen.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

`Warning: DEACTIVATE has no effect`: The `$deactivate` routine has no effect when using this kernel.

`Warning: IGNORE has no effect`: The `$ignore` routine has no effect when using this kernel.

`Warning: NULL_OUTPUT ignored`: The `$nulloutput` routine has no effect when using this kernel.

`Warning: process ... no action for: ...`: An event combination has occurred for which there is no action specified.

`Warning: RelinquishProcessor: has no effect`: This should never happen.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Bibliography

Agre, J. R., ‘‘Simulation of Time Warp Distributed Simulations,’’ pp. 85-90 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Apollo, ‘‘Network Computing System (NCS) Reference,’’ Order No. 010200, Apollo Computer Inc., Chelmsford, Massachusetts, 1987.

Ayani, R., ‘‘A Parallel Simulation Scheme Based on Distances Between Objects,’’ pp. 113-118 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Baezner, D., J. Cleary, G. Lomow, and B. W. Unger, ‘‘Algorithmic Optimizations on Time Warp,’’ pp. 73-78 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Bailey, M. L. and L. Snyder, ‘‘The Effect of Timing on the Parallelism Available for Parallel Circuit Simulation,’’ pp. 157-163 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Brown, R., ‘‘Calendar Queues: A Fast O(1) Priority Queue Implementation of the Simulation Event Set Problem,’’ *Communications of the ACM*, Vol. 31, No. 10, pp. 1220-1227, October 1988.

Chandy, K. M. and J. Misra, ‘‘Distributed Simulation: A Case Study in Design and Verification of Distributed Programs,’’ *IEEE Trans. on Software Engineering*, Vol. SE-5, No. 5, pp. 440-452, September 1979.

Chandy, K. M. and J. Misra, ‘‘Conditional Knowledge as a Basis for Distributed Simulation,’’ Technical Report 5251:TR:87, Computer Science Department, California Institute of Technology, Pasadena, California, 1987.

Ebling, M., M. Di Loreto, M. Presley, F. Wieland, and D. Jefferson, ‘‘An Ant Foraging Model Implemented on the Time Warp Operating System,’’ pp. 21-26 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Fujimoto, R. M., ‘‘Performance Measurements of Distributed Simulation Strategies,’’ Technical Report No. UUCS-87-026a, Computer Science Department, University of Utah, Salt Lake City, Utah, 1987.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Fujimoto, R. M., ‘‘Performance Measurements of Distributed Simulation Strategies,’’ pp. 14-20 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and D. Jefferson, Society for Computer Simulation, San Diego, California (February 1988).

Fujimoto, R. M., ‘‘Lookahead in Parallel Discrete Event Simulation,’’ pp. 34-41 in *Proc. 1988 Int. Conf. on Parallel Processing*, , St Charles, Illinois (August 1988).

Fujimoto, R. M., ‘‘Performance Measurements of Distributed Simulation Strategies,’’ *Trans. of the Society for Computer Simulation*, Vol. 6, No. 2, pp. 89-132, April 1989.

Fujimoto, R. M., ‘‘Time Warp on a Shared Memory Multiprocessor,’’ *Trans. of the Society for Computer Simulation*, Vol. 6, No. 3, pp. 211-239, July 1989.

Fujimoto, R. M., ‘‘Time Warp on a Shared Memory Multiprocessor,’’ pp. 242-249 in *Proc. 1989 Int. Conf. on Parallel Processing*, Computer Science Department, University of Utah, St Charles, Illinois (August 1989).

Fujimoto, R. M., ‘‘Performance of Time Warp Under Synthetic Workloads,’’ pp. 23-28 in *Proc. SCS Western Multiconf. — Distributed Simulation*, Society for Computer Simulation, San Diego California. (January 1990).

Hontalas, P., B. Beckman, M. Di Loreto, L. Blume, P. Reiher, K. Sturdevant, L. Van Warren, J. Wedel, F. Wieland, and D. Jefferson, ‘‘Performance of the Colliding Pucks Simulation on the Time Warp Operating Systems (Part 1: Asynchronous Behavior and Sectoring),’’ pp. 3-7 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Jager, W. J., ‘‘The P-MACHine: A Hardware Message Accelerator for a Multiprocessor System,’’ M.A.Sc. Thesis, CCNG Technical Report T-165, University of Waterloo, Department of Electrical Engineering, Waterloo, Ontario, 1987.

Jager, W. J. and W. M. Loucks, ‘‘The P-MACHine: A Hardware Message Accelerator for a Multiprocessor System,’’ *Proc. 1987 Int. Conf. on Parallel Processing*, pp. 600-609, Pennsylvania State University Press, 1987.

Jefferson, D., B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, ‘‘Distributed Simulation and the Time Warp Operating System,’’ pp. 77-93 in *Proc. 12th SIGOPS — Symposium on Operating Systems Principles*, (1987).

Jefferson, D. R., ‘‘Virtual Time,’’ *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, July 1985.

Johnson, S. C., ‘‘Yacc — Yet Another Compiler Compiler,’’ in *Unix Programmer’s Manual*, Bell Laboratories, Murray Hill, NJ (1979).

Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Kleinrock, L., *Queueing Systems*, New York, NY: John Wiley & Sons, 1975.

Lesk, M. E., ‘‘Lex — A Lexical Analyzer Generator,’’ in *Unix Programmer’s Manual*, Bell Laboratories, Murray Hill, NJ (1979).

Leung, E., J. Cleary, G. Lomow, D. Baezner, and B. Unger, ‘‘The Effects of Feedback on the Performance of Conservative Algorithms,’’ pp. 44-49 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Loucks, W. M. and Z. G. Vranesic, ‘‘FERMTOR — A Flexible Extendible Range Multiprocessor,’’ *CIPS Session 80*, pp. 134-149, Canadian Information Processing Society, 1980.

Loucks, W. M. and B. R. Preiss, ‘‘The Role of Knowledge in Distributed Simulation,’’ pp. 9-16 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, Society for Computer Simulation (January 1990).

Lubachevsky, B. D., ‘‘Bounded Lag Distributed Discrete Event Simulation,’’ pp. 183-191 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and D. Jefferson, Society for Computer Simulation, San Diego, California (February 1988).

Lubachevsky, B. D., ‘‘Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks,’’ *Communications of the ACM*, Vol. 32, No. 1, pp. 111-123, January 1989.

Lubachevsky, B. D., ‘‘Scalability of the Bounded Lag Distributed Discrete Event Simulation,’’ pp. 100-107 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Misra, J., ‘‘Distributed Discrete-Event Simulation,’’ *ACM Computing Surveys*, Vol. 18, No. 1, pp. 39-66, March 1986.

National Semiconductor, *Logic Databook*, Santa Clara, California: National Semiconductor Corp., 1981.

Nicol, D. M., ‘‘High Performance Parallelized Discrete Event Simulation of Stochastic Queueing Networks,’’ pp. 306-314 in *Proc. 1988 Winter Simulation Conf.*, ed. M. A. Abrams, P. L. Haigh, and J. C. Comfort, Society for Computer Simulation, San Diego, California (December 1988).

Preiss, B. R., W. M. Loucks, and V. C. Hamacher, ‘‘A Unified Modeling Methodology for Performance Evaluation of Distributed Discrete Event Simulation Mechanisms,’’ pp. 315-324 in *Proc. 1988 Winter Simulation Conf.*, ed. M. A. Abrams, P. L. Haigh, and J. C. Comfort, Society for Computer Simulation, San Diego, California (December 1988).

Preiss, B. R. and W. M. Loucks, ‘‘Prediction and Lookahead in Distributed Simulation,’’ CCNG Technical Report E-191, Computer Communications Networks Group, University of Waterloo, Waterloo, Ontario, Canada, 1989.

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Preiss, B. R., ‘‘The Yaddes Distributed Discrete Event Simulation Specification Language and Execution Environments,’’ pp. 139-144 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Preiss, B. R., ‘‘Performance of Discrete Event Simulation on a Multiprocessor Using Optimistic and Conservative Synchronization,’’ pp. 5 in *Proc. 1990 Int. Conf. on Parallel Processing*, , St Charles, Illinois (August 1990).

Preiss, B. R. and W. M. Loucks, ‘‘The Impact of Lookahead on the Performance of Conservative Distributed Simulation,’’ pp. 6 in *Proc. SCS European Multiconf. — Simulation Methodologies, Languages and Architectures*, ed. ?, Society for Computer Simulation, Nuremberg, FRG (June 1990).

Presley, M., M. Ebling, F. Wieland, and D. Jefferson, ‘‘Benchmarking the Time warp Operating System with a Computer Network Simulation,’’ pp. 8-13 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Reed, D. A., A. D. Malony, and B. D. McCredie, ‘‘Parallel Discrete Event Simulation Using Shared Memory,’’ *IEEE Trans. on Software Engineering*, Vol. 14, No. 4, pp. 541-553, April 1988.

Rose, J., W. Loucks, and Z. Vranesic, ‘‘FERMTOR — A Tuneable Multiprocessor Architecture,’’ *IEEE Micro*, Vol. 5, No. 4, pp. 5-17, 1985.

Su, W.-K. and C. L. Seitz, ‘‘Variants of the Chandy-Misra-Bryant Distributed Discrete Simulation Algorithm,’’ pp. 38-43 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Wagner, D. B. and E. D. Lazowska, ‘‘Parallel Simulation of Queueing Networks: Limitations and Potentials,’’ *Performance Evaluation Review*, Vol. 17, No. 1, pp. 146-155, May 1989.

Wieland, F., L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, ‘‘Distributed Combat Simulation and Time Warp: The Model and Its Performance,’’ pp. 14-20 in *Proc. SCS Eastern Multiconf. — Distributed Simulation*, ed. B. W. Unger and R. M. Fujimoto, Society for Computer Simulation, Tampa, Florida (March 1989).

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Index

#....26
#define....65
#ifdef....65
#include....48,65
$....20,22,26
$activate....22,23,31,38,68,74,76,77
$deactivate....15,23,31,38,65,68,73,74,78
$event....21,63
$eventoccurred....21,23
$ignore....15,24,31,38,65,68,73,75,76,78
$name....21
$nulloutput....24,25,31,37,38,55,56,57,58,65,68,78
$output....24,25,31,37,38,55,56,57,58,68
$pid....22
$settimelimit....25,55,56,75,76
$state....22,63
$time....22,63
%....26
*....26
*/....26
+....26
+a....30
+h....30,68
+i....31,68
+k....31,68
+m....31,68
+nmc....31
+o....31
+p....31,68
+r....68
+s....31,68
+t....31,68
-....26
-h....30
-w....31,68
.b4....49,50
.bin....48
.c....27,68

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

.c4x....48
.d4x....49
.y....27,68
/ 26
/* 26
AccumulateStatistic 35
action....14,15,16,63
and....14
anything 15
argc....20
argv 20
cc....51,66,68
chandy-misra.a....73
CleanUp 20
cm....48
config 49
configc....49
connect 18
DeclareStatistic 34
default 16
DefaultFinal 20,36
DefaultInitial....19
dispatch 51,66,67,68,69
DispatchProcesses....64,73,76
dist-chandy-misra.a....66
dist-multi-list.a....66
dist-virtual-time.a....66
el....48
EPSILON 39
event-list.a....74
final 14,63
FinalProcedure 64
flagman 51,52,66,67,68,69
fprintf 48
hardwire 49
iboot 49
ilink 48
initial 13,14,63,9
InitializeRNG....32
InitializeStatistic 73,74,75,77
InitialProcedure....64
inputs....11

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Inputs 62
iserver....50
LaunchProcess 64
ld 66,68
main 48
ml 48
mms2....49
mrclean....52,66,67,69
multi-list.a 75
node....16
none....11,12
not 14
nothing 15,63,65
on....17,64
or 14,15
outputs....12
Outputs....63
POPULATION....34
postamble 10
preamble....10
printf....48
process....16,63,64
Random....33
ReadFromFile 9
ReportStatistic....35,36
RNG_INITIAL_STATE 33
RNG_INITIAL_VALUE....32
RNG_TYPE 32
sacrtl.lib....48
SetUp....19
slave.b4 49
softwire 49
something....15
StartTasks....76
state 12,13,22
STATE 22,63
StateTable 63
STATISTIC_INITIAL_STATE....35
STATISTIC_INITIAL_VALUE 34
STATISTIC_TYPE....34
stdio.h....48
stdout....48

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

struct....12,13
t4c....48
taskharn.t4x....48
TIME_AVERAGED....34
TwoInputNand....22,9
virtual-time.a....76
vt....48
WriteToFile....9
yaddes.cfg....49

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

Table of Contents

Chapter 1. Introduction 2

Chapter 2. Computer Simulation and Modelling 3

Chapter 3. Synchronization Methods 4

3.1. Sequential Simulation 4

3.2. Distributed Simulation using Multiple Event Lists 4

3.3. Conservative Distributed Simulation 5

3.4. Optimistic Distributed Simulation 5

Chapter 4. Example: Exclusive-OR Circuit 6

Chapter 5. The Yaddes Specification Language 10

5.1. Model Specifications 10

5.1.1. Inputs 11

5.1.2. Outputs 12

5.1.3. State 12

5.1.4. Initial State 13

5.1.5. Actions 13

5.1.5.1. General Event Combination Actions 13

5.1.5.2. Initial Action 14

5.1.5.3. Final Action 14

5.1.5.4. No Event Action 14

5.1.5.5. Some Event Action 15

5.1.5.6. Any Event Action 15

5.1.5.7. Default Action 15

5.2. Process Specifications 16

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.3. Connection Specifications 17

5.4. Procedure Specifications 19

5.4.1. Initial procedure 19

5.4.2. Final procedure 19

5.5. Predefined Variables and Constants 20

5.5.1. Constants 20

5.5.2. Variables 20

5.5.2.1. `$event` 20

5.5.2.2. `$name` 21

5.5.2.3. `$pid` 21

5.5.2.4. `$state` 21

5.5.2.5. `$time` 22

5.6. Predefined Functions 22

5.6.1. `$activate` 22

5.6.2. `$deactivate` 22

5.6.3. `$eventoccurred` 23

5.6.4. `$ignore` 23

5.6.5. `$output` 23

5.6.6. `$nulloutput` 24

5.6.7. `$settimelimit` 24

5.7. Constants and Manifest Expressions 25

5.8. Comments 26

5.9. Compiling and Running the Simulation 26

5.9.1. Compiling a Yaddes Language Specification 26

5.9.1.1. Separate Compilation 26

5.9.1.2. Synchronization Method Libraries 27

5.9.2. Running a Simulation Program 29

5.9.2.1. `+a` 29

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

5.9.2.2. `+h` **and** `-h` 29
5.9.2.3. `+i` 29
5.9.2.4. `+k` 29
5.9.2.5. `+m` 29
5.9.2.6. `+o` 29
5.9.2.7. `+p` 30
5.9.2.8. `+s` 30
5.9.2.9. `+t` 30
5.9.2.10. `-w` 30
5.9.2.11. `+nmc` 30
Chapter 6. Pseudo-Random Number Generator Package 31
6.1. Using the Pseudo-Random Number Generator Package 31
Chapter 7. Statistics Gathering and Reporting Package 33
7.1. Distributed Statistics Collection Capability 35
Chapter 8. Avoiding deadlock when using Conservative Synchronization 36
8.1. Using `$output` **and** `$nulloutput` 36
8.2. Using `$ignore` 37
8.3. Using `$activate` **and** `$deactivate` 37
Chapter 9. Example: Queue and Server 38
9.1. The Arrival LP 38
9.2. The Queue LP 40
9.3. The Server LP 43
9.4. The complete specification 44
Appendix A. Distributed Simulation on a Transputer Network 47
A.1. Preparing the Yaddes Source File 47

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

A.2. Compiling the C program file 47
A.3. Linking the program 47
A.4. Configuring the file for execution 48
A.5. Configuring the Transputer Network 48
A.6. Running the Simulation 49
Appendix B. Distributed Simulation on an Apollo Network 50
B.1. Creating the Executable File 50
B.2. Running the Distributed Simulation 50
B.3. Simulation Cleanup 51
Appendix C. Yaddes BNF 52
Appendix D. Kernel Statistics 54
D.1. Libraries for Sequential Simulation 54
D.2. Libraries for Distributed Simulation using Multiple Event Lists 54
D.3. Libraries for Conservative Simulation 56
D.4. Libraries for Optimistic Simulation 56
Appendix E. Yaddes Compiler Output 58
Appendix F. Pitfalls 64
Appendix G. Yaddes `man` Pages 65
G.1. `dispatch` 65
NAME 65
SYNOPSIS 65
DESCRIPTION 65
SEE ALSO 65
G.2. `flagman` 65
NAME 65

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved

SYNOPSIS 66
DESCRIPTION 66
SEE ALSO 66

G.3. `mrclean` 66
NAME 66
SYNOPSIS 66
DESCRIPTION 66
SEE ALSO 66

G.4. `yaddes` 67
NAME 67
SYNOPSIS 67
DESCRIPTION 67
SIMULATION OPTIONS 67
FILES 67
SEE ALSO 68

Appendix H. Yaddes Compile-Time Error Messages 69

Appendix I. Run-Time Error Messages 72

I.1. `chandy-misra.a`**:** 72

I.2. `event-list.a`**:** 73

I.3. `multi-list.a`**:** 74

I.4. `virtual-time.a`**:** 75

Bibliography 78

Index 82

Copyright (c) 1990 by the Computer Communications Networks Research Group, University of Waterloo. All rights reserved